Кириченко А. В.

JavaScript

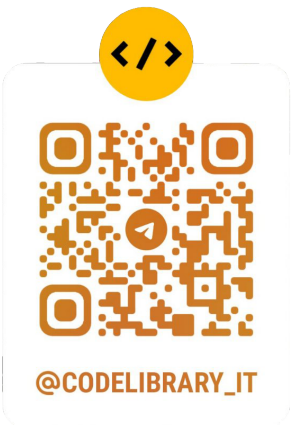
для FrontEnd-разработчиков

Написание. Тестирование. Развертывание

Кириченко А. В.

JavaScript для FrontEnd-разработчиков

Написание. Тестирование. Развертывание



УДК 681.3.068; 004.738

ББК 32.973

ISBN 978-5-94387-789-6

Кириченко А. В.

JavaScript для FrontEnd-разработчиков. Написание. Тестирование. Развертывание. — СПб.: Наука и Техника, 2020. — 320 с.: ил.

Данная книга посвящена тому, как на языке JavaScript создавать хороший код для фронтенда (и не только). В книге последовательно затронуты все аспекты производства JavaScript-кода: от выбора архитектуры и конструирования кода до покрытия модульными тестами, отладки, интеграционного тестирования, сборки и непрерывной поставки вашего кода. Рассматриваются как общие моменты – постановка процесса разработки, событийно-ориентированная архитектура JavaScript-приложений, техника непрерывной интеграции, так и предельно конкретные вопросы – как и какие инструменты (фреймворки) использовать для той или иной задачи, что конкретное нужно делать в том или ином случае, какие ошибки встречаются. Попутно в книге рассмотрено применение большого количества инструментов. Существенное внимание уделено автоматизации на всех этапах создания и поставки JavaScript-кода.

Книга написана доступным языком и представляет несомненный интерес для всех, кто занимается или планирует заняться программированием на JavaScript, хочет повысить качество своего JavaScript-кода, добиться высокой эффективности в создании качественного кода фронтенда. Книга будет полезна как начинающим, так и опытным JavaScript-разработчикам.

ISBN 978-5-94387-789-6

9 78- 5- 94387- 789- 6

Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru
192029, г. Санкт-Петербург,

© Кириченко А. В.
© Наука и техника (оригинал-макет)

Содержание

Глава 1. Хороший код фронтенда на JavaScript — что это? 11

1.1. Небольшая прелюдия о процессах разработки 12

ГИБКАЯ МЕТОДОЛОГИЯ РАЗРАБОТКИ 12

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ. TEST-DRIVEN DEVELOPMENT (TDD) 16

УПРАВЛЯЕМАЯ ПОВЕДЕНИЕМ РАЗРАБОТКА 17

КАКОЙ ПОДХОД ЛУЧШЕ? 18

1.2. Код для людей 19

1.2.1. НЕСКОЛЬКО ВОПРОСОВ «ПОЧЕМУ?» 20

ПОЧЕМУ ХОРОШИЙ КОД — ЭТО КОД ТЕСТИРУЕМЫЙ? 20

ПОЧЕМУ ХОРОШИЙ КОД ФРОНТЕНДА – ЭТО КОД УДОБНЫЙ В СОПРОВОЖДЕНИИ? 21

ПОЧЕМУ ХОРОШИЙ КОД ФРОНТЕНДА – ЭТО КОД ПОНЯТНЫЙ? 22

1.2.2. НЕСКОЛЬКО ВОПРОСОВ «КАК?» 23

КАК СДЕЛАТЬ КОД ТЕСТИРУЕМЫМ? СВЯЗНОСТЬ КОДА 23

КАК СДЕЛАТЬ КОД УДОБНЫМ В СОПРОВОЖДЕНИИ? 23

КАК СДЕЛАТЬ КОД ПОНЯТНЫМ? 24

1.3. Что еще обеспечивает идеальность кода? 24

ТЕСТИРОВАНИЕ 25

ОТЛАДКА 25

Резюме 26

Глава 2. Сложность кода 27

2.1. Размер кода как характеристика сложности 29

РАЗМЕР КОДА 29

МИНИМИЗАЦИЯ РАЗМЕРА КОДА 30

2.2. Статический анализатор кода JSLint 38

2.3. Цикломатическая сложность 44

2.4. Повторное использование кода 48

2.5. Разветвление на выходе (Fan-Out) 53

2.6. Разветвление на входе 64

2.7. Связанность 66

СВЯЗАННОСТЬ НА УРОВНЕ СОДЕРЖИМОГО 66

ОБЩАЯ СВЯЗАННОСТЬ 67

СВЯЗАННОСТЬ НА УРОВНЕ УПРАВЛЕНИЯ 67

СВЯЗАННОСТЬ ПО ОТПЕЧАТКУ В СТРУКТУРЕ ДАННЫХ (STAMP COUPLING) 67

СВЯЗАННОСТЬ НА УРОВНЕ ДАННЫХ 68

НЕТ СВЯЗАННОСТИ 68

ИНСТАНЦИРОВАНИЕ 68

МЕТРИКИ СВЯЗАННОСТИ 69

СВЯЗАННОСТЬ В РЕАЛЬНОМ МИРЕ 70

ТЕСТИРОВАНИЕ СВЯЗАННОГО КОДА 73

ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ 74

2.8. Комментарии 78

ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТА YUIDOC ДЛЯ ГЕНЕРАЦИИ ДОКУМЕНТАЦИИ 80

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ ГЕНЕРАЦИИ ДОКУМЕНТАЦИИ С ПОМОЩЬЮ ИНСТРУМЕНТА JSDOC 82

ГЕНЕРАТОРЫ ДОКУМЕНТАЦИИ DOCCO/ROCCO 84

2.9. Тестирование человеком: человеческий тест, размер кода и инспекция кода по Фагану 86

Резюме 88

Глава 3. Событийно-ориентированная архитектура 89

3.1. Преимущества событийно-ориентированного программирования 90

3.2. Концентратор событий 92

ЧТО ТАКОЕ КОНЦЕНТРАТОР СОБЫТИЙ 92

ИСПОЛЬЗОВАНИЕ КОНЦЕНТРАТОРА СОБЫТИЙ 95

ОТВЕТЫ НА ПОРОЖДЕННЫЕ СОБЫТИЯ 101

3.3. Контекст применения событийно-ориентированной архитектуры 103

СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА И MVC-МЕТОДЫ 103

СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ 104

СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА И SAAS 105

3.4. Веб-ориентированные приложения 105

3.5. Тестирование событийно-ориентированной архитектуры 106

3.6. Недостатки и "узкие" места событийно-ориентированной архитектуры 112

МАСШТАБИРУЕМОСТЬ 112

ШИРОКОВЕЩАНИЕ 112

ПРОВЕРКА ВРЕМЕНИ ВЫПОЛНЕНИЯ 113

БЕЗОПАСНОСТЬ 113

СОСТОЯНИЕ СЕССИИ 114

3.7. Интеллектуальный концентратор: коммутатор событий 114

РАЗВЕРТЫВАНИЕ 115

ОДНОАДРЕСНЫЕ СОБЫТИЯ 116

ШИРОКОВЕЩАТЕЛЬНЫЕ СОБЫТИЯ 117

РЕАЛИЗАЦИЯ 118

СЕССИИ (СЕАНСЫ) 121

РАСШИРЯЕМОСТЬ 122

Глава 4. Модульное тестирование 123

4.1. Выбор фреймворка (тестовой платформы) 124

4.2. Давайте приступим 126

4.3. Пишем совершенные тесты 128

ИЗОЛЯЦИЯ 129

ОБЛАСТЬ ДЕЙСТВИЯ ТЕСТА, ГРАНИЦЫ ТЕСТИРОВАНИЯ 130

4.4. Определение ваших функций 130

БЛОКИ КОММЕНТАРИЕВ 130

ТЕСТЫ КАК ПЕРВООСНОВА 132

4.5. Подходы в тестировании 132

ПОЗИТИВНОЕ ТЕСТИРОВАНИЕ 132

НЕГАТИВНОЕ ТЕСТИРОВАНИЕ 133

АНАЛИЗ ПОКРЫТИЯ КОДА 134

4.6. Практическое тестирование 135

4.6.1. РАБОТА С ЗАВИСИМОСТЯМИ 135

ДУБЛИ 135

МОСК-ОБЪЕКТЫ (ИМИТАЦИИ) 136

ЗАГЛУШКИ 137

ШПИОНЫ 138

4.7. Асинхронное тестирование 139

ИСПОЛЬЗОВАНИЕ YUI TEST 140

4.8. Запуск тестов: сторона клиента 141

4.8.1. PHANTOMJS 142

4.8.2. SELENIUM 148

ТЕСТИРОВАНИЕ В РЕАЛЬНЫХ БРАУЗЕРАХ 148

МОБИЛЬНОЕ ТЕСТИРОВАНИЕ ... 155
IOS ... 155
ANDROID ... 156

4.9. Тестирование: сторона сервера ... 157

4.9.1. ФРЕЙМВОРК JASMINE ... 157

ЗАВИСИМОСТИ ... 159
ШПИОНЫ ... 162
ВЫВОД РЕЗУЛЬТАТОВ ТЕСТИРОВАНИЯ ... 165

Резюме ... 166

Глава 5. Покрытие кода ... 167

5.1. Основы покрытия кода ... 168

5.2. Данные покрытия кода ... 172

5.3. Практика определения покрытия кода ... 173

ИНСТРУМЕНТИРОВАНИЕ ФАЙЛОВ ... 173
АНАТОМИЯ ФАЙЛА ПОКРЫТИЯ ... 174

5.4. Развертывание ... 176

РАЗВЕРТЫВАНИЕ JAVASCRIPT-КЛИЕНТА ... 176
РАЗВЕРТЫВАНИЕ JAVASCRIPT СЕРВЕРА ... 178

5.5. Сохранение информации о покрытии ... 183

В СЛУЧАЕ МОДУЛЬНОГО ТЕСТИРОВАНИЯ ... 183
В СЛУЧАЕ ИНТЕГРАЦИОННОГО ТЕСТИРОВАНИЯ ... 185

5.6. Генерация вывода ... 186

5.7. Агрегация ... 187

5.8. Скрытые файлы ... 189

5.9. Цели покрытия ... 193

Глава 6. Интеграционное, нагрузочное и тестирование производительности 195

6.1. Важность интеграции 196

6.2. Интеграционное тестирование. Selenium 197

СЕРИЯ ПРОДУКТОВ SELENIUM 197

WEBDRIVERJS И НАПИСАНИЕ ИНТЕГРАЦИОННЫХ ТЕСТОВ НА JAVASCRIPT 201

SELENIUM REMOTE CONTROL 204

SELENIUM GRID 206

6.3. CasperJS 207

6.4. Тестирование производительности 211

СОЗДАНИЕ HAR-ФАЙЛОВ 211

ИСПОЛЬЗОВАНИЕ ПРОКСИ 212

ПРОСМОТР HAR-ФАЙЛОВ 217

ТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ БРАУЗЕРА 220

6.5. Нагрузочное тестирование 222

НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ БРАУЗЕРА 223

6.6. Отслеживание использования ресурсов 229

6.6.1. ОТСЛЕЖИВАНИЕ ЗАДЕЙСТВОВАНИЯ РЕСУРСОВ НА СТОРОНЕ КЛИЕНТА 231

ИСПОЛЬЗОВАНИЕ ПАМЯТИ 231

ИСПОЛЬЗОВАНИЕ ПРОЦЕССОРА 235

Глава 7. Отладка. "Допиливаем" наш фронтенд 237

7.1. Отладка средствами браузера 238

ОТЛАДЧИК FIREFOX 240

ОТЛАДЧИК CHROME 243

ОТЛАДЧИК SAFARI 247

ОТЛАДЧИК INTERNET EXPLORER И EDGE 249

7.2. Отладка Node.js 250

7.3. Удаленная отладка 253

УДАЛЕННАЯ ОТЛАДКА В ОКРУЖЕНИИ CHROME 253

ОТЛАДКА В PHANTOMJS 260

ОТЛАДКА В FIREFOX 261

7.4. Мобильная отладка 262

ANDROID 262

IOS 264

КРОССПЛАТФОРМЕННОЕ РЕШЕНИЕ ADOBE EDGE INSPECT 265

ДРУГИЕ ВОЗМОЖНОСТИ МОБИЛЬНОЙ ОТЛАДКИ. WINDOWS PHONE 269

7.5. Производственная отладка 270

МИНИФИКАЦИЯ И ДЕМИНИФИКАЦИЯ КОДА 270

КАРТЫ КОДА 271

Глава 8. Автоматизация и развертывание 277

8.1. Что автоматизировать? 278

8.2. Когда автоматизировать? 279

8.3. Как автоматизировать? 279

8.3.1. АВТОМАТИЗАЦИЯ С НЕПРЕРЫВНОЙ ИНТЕГРАЦИЕЙ 280

8.3.2. АВТОМАТИЗАЦИЯ РАЗРАБОТКИ 281

РЕДАКТОР КОДА 282

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ ПО ХОДУ РАЗРАБОТКИ КОДА 282

ПРОСМОТР КОДА. КАК АВТОМАТИЗИРОВАТЬ? 288

ЛОВУШКИ ПРЕДВАРИТЕЛЬНОЙ ФИКСАЦИИ (КОММИТОВ) 288

ДРУГИЕ ИНСТРУМЕНТЫ РАЗРАБОТЧИКА 292

8.3.3. АВТОМАТИЗАЦИЯ СРЕДЫ СБОРКИ 293

СБОРКА 293

JENKINS 294

СОЗДАНИЕ ПРОЕКТА JENKINS 296

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ СРЕДСТВАМИ SELENIUM 300

ВЫВОД МОДУЛЬНОГО ТЕСТА 303

ВЫВОД ПОКРЫТИЯ КОДА 304

СЛОЖНОСТЬ 307

JSLINT + JENKINS 308

ДУБЛИРУЮЩИЙСЯ КОД 310

УВЕДОМЛЕНИЯ 314

ЗАВИСИМЫЕ СБОРКИ 314

НИЧЕГО ЛИШНЕГО 314

РАБОТАЕМ БЕЗ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ 315

8.3.4. РАЗВЕРТЫВАНИЕ 316

Резюме 318

Глава 1.

Хороший код фронтенда на JavaScript — что это?

1.1. Небольшая прелюдия о процессах разработки

На фоне ручной и крайне индивидуалистичной разработки программного обеспечения было предпринято множество попыток стандартизировать всю эту «разработчиковскую» кухню. Делалось и делается это для того, чтобы процесс и практики написания «хорошего» кода сделать более повторяемыми и регламентированными. Чтобы, с одной стороны, не приходилось каждый раз изобретать велосипед, а с другой – повысить вероятность того, что проект будет успешным, код «идеальным», а все – счастливы. И если с рецептами счастья как-то все непросто, то в остальном определенные успехи есть.

Поскольку цель данной книги состоит не только в том, чтобы рассмотреть те или иные методики и дать контекст их применимости для достижения «идеальности» кода, но и в том, чтобы попрактиковаться в программистском думании (что на «идеальность» кода оказывает едва ли не бОльшее влияние), то давайте в начале книги поговорим о некоторых текущих взглядах на процесс разработки программного обеспечения, которые будут нам полезны по ходу дела.

Гибкая методология разработки

Agile – это одна из самых универсальных практик. Гибкий подход[1] является, главным образом, ответом на водопадную модель разра-

1 Agile – в пер. с англ. – гибкий.

ботки (она же каскадная модель). При этом под водопадной моделью принято понимать модель процесса разработки программного обеспечения, в которой процесс разработки выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.

В свою очередь гибкая методология — это семейство подходов к разработке программного обеспечения, ориентированных на использование итеративной разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп-команд, состоящих из специалистов разного профиля.

В водопадной модели каждый шаг в процессе разработки происходит последовательно и в изоляции. Следующий этап не начинается, пока не закончится предыдущий. Тестеры ожидают, пока кодеры напишут код, затем все ожидают, пока тестеры протестируют написанный кодерами код и т.д.

Гибкая (Agile) разработка пытается быть более адаптивной и изменяемой, позволяет каждому этапу разработки происходить параллельно, а не последовательно. При этом что-то работающее обладает гораздо большим приоритетом, чем теоретические измышления и проектные решения. Просто потому, что на основе этого чего-то можно уже получить обратную связь, и либо двигаться дальше, либо что-то переделать, не затратив на это много усилий. Поэтому в agile-методиках стремятся как можно быстрее что-то сделать/показать заказчику, а потом уже наращивать/развивать решение, что-то добавляя, изменяя, развивая и т.д.

Большие блоки работы разбиваются на меньшие блоки, так как меньшие блоки удобнее контролировать и оценивать. Вместо того чтобы месяцами ждать завершения одного большого блока, на каждом из этапов выполняется свой небольшой блок работы, и затем за нее берется другая команда, в итоге работа над проектом ведется всегда, а время достижения конечной цели может быть существенно сокращено. Кроме того, за счет постоянного взаимодействия с заказчиком (анализ требований ведется постоянно, заказчику постоянно показываются и сдаются все новые и новые улучшения, добавления, доработки и проч.) повышается вероятность того, что созданный продукт будет максимально удовлетворять потребностям

заказчика. Это все в теории, на практике бывает по-разному, но это тема отдельной книги.

Обратите внимание, что использование гибкой методологии не обязательно означает, что работа над приложением будет завершена быстрее или с более высоким качеством. Самое большое преимущество этой методологии — способ, которым она обрабатывает изменения. В водопадной модели любое изменение требует прогона по всему процессу заново. В более коротких циклах гибкой модели изменения очень легко включить в конечный продукт.

Практически все современные программирующие фирмы в той или иной степени используют гибкие подходы разработки: кто-то больше (внедряя у себя Scrum, например), кто-то меньше – пользуясь той или иной отдельной методикой или подходом. Вполне вероятно, что и у вас в компании (если вы работаете в какой-то компании) практикуются гибкие подходы. Однозначно судить об этом вы можете, если слышите такие слова, как бэклог (backlog), стендапмитинг (standup), непрерывная поставка (или любое словосочетание со словом continuous), fail-fast и т.п.

Примечание

Рассмотрение, что есть что в гибкой разработке, и что означают вышеприведенные слова, выходит за рамки данной книги. Сделать это вы можете на страницах соответствующей литературы и Интернета. Но есть одно словосочетание, которое, с одной стороны, может вызвать сложность, а с другой стороны – имеет непосредственное отношение к качеству кода. Поэтому давайте поясним его прямо здесь. Словосочетание это — fail-fast, а означает оно, что программа сигнализирует об ошибке практически сразу же после возникновения этой ошибки в программе, а не потом, когда уже очень трудно разобраться, где именно произошла ошибка.

На рис. 1.2 показана каноническая диаграмма процесса гибкой разработки.

Основная идея схемы на рис. 1.2 заключается в том, что быстрая итерация и постоянное взаимодействие команд ускоряют поставку качественного программного обеспечения.

Гибкая разработка не регламентирует, как должно быть написано программное обеспечение. Скорее, она рекомендует несколько под-

ходов, соответствующих Agile-философии. Например, требования в Agile принято формулировать в виде "пользовательских историй", то есть в формате того, что кто-то, как пользователь, хочет что-то получить от использования разрабатываемого вами ПО. Эти "истории" расцениваются как запросы новых функций для вашего приложения. Пользователь – это любой, кто использует ваше приложение: домашний пользователь, разработчик, сторонняя система, API и т.д.

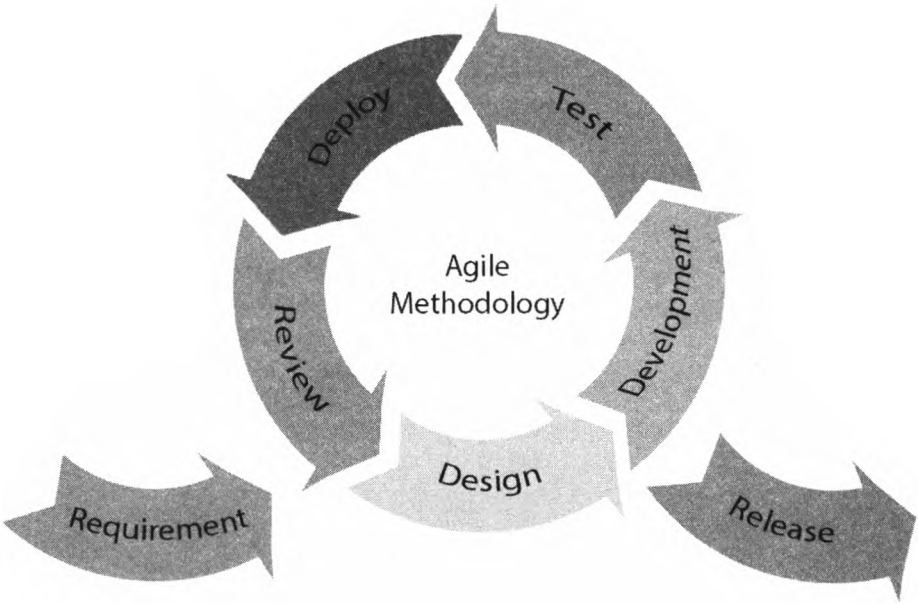


Рис. 1.2. Гибкая разработка

Или еще одна методология разработки, связанная с гибкой разработкой, – парное программирование. Самая простая и «чистая» форма парного программиста выглядит так: два программиста, уставившись в один и тот же монитор, используя одну клавиатуру и мышь, пишут программу вместе. Пока один программист вводит код, другой активно его на лету отлаживает, ищет ошибки. Две головы лучше, чем одна, – как говорят про пограничника с собакой, – поэтому возможные решения будут найдены быстрее, чем если бы эти два программиста работали по отдельности.

Разработка через тестирование. Test-Driven Development (TDD)

Разработка через тестирование (TDD) — рекомендуемая практика гибкой разработки программного обеспечения. TDD требует, чтобы вы сначала писали тесты, а затем уже приступали к написанию кода.

Поскольку вы сначала пишете тест, а затем уже код, у вас никогда не будет непротестированного кода. Тесты призваны гарантировать, что код, после его написания, будет соответствовать вашим ожиданиям. А соответствие полученного результата изначальным ожиданиям – это главное качество и гарантия успешности разработки. По крайней мере, так в теории. На практике же разработчики не всегда пишут тесты для всего предполагаемого кода, даже когда они целенаправленно придерживаются практики TDD. Но даже благодаря частичному предварительному покрытию тестами хотя бы часть кода будет гарантированно протестированной, что есть полезно, а во многих случаях и критически важно.

Применимость TDD обусловлена особенностями проекта и здравым смыслом. TDD работает лучше всего тогда, когда вы начинаете новый проект или разработку нового модуля. Особенно TDD успешна, когда вам требуется только модульное тестирование. В то же время, написание полных интеграционных тестов для всего проекта перед самим проектом выглядит пугающе!

TDD может быть серьезным основанием или причиной, по которой будет принято решение заново переписать унаследованный код. Если разработчик станет перед выбором "покрыть тестами уже существующий унаследованный код" или "написать свой код, сначала разработав для него тесты", вероятнее всего, он выберет последний вариант. Конечно, у разработчиков далеко не всегда есть выбор, но в любом случае следует иметь в виду, что писать тесты к уже существующему коду бывает ох как непросто (а иногда и нереально), и в любом случае гораздо сложнее, чем это делать заранее. Кроме того, когда в самом начале еще нет кода, "стоимость" написания тестов – минимальна, так как не тратится время на анализ и понимание самого кода.

В 2005 году исследование канадских студентов установило, что TDD сделало программистов более продуктивными и эффективными за счет того, что они писали больше тестов. При этом исследователи отметили, что качество кода линейно увеличивается в соответствии с числом проведенных тестов независимо от используемой стратегии. То есть число тестов пропорционально более высокому качеству кода.

С какими-то деталями данного исследования можно бы и поспорить, но даже на уровне здравого смысла и практического опыта можно сделать следующее утверждение: любая методология, которая заставляет, обязывает разработчиков писать больше тестов до, во время или после написания кода, уже хороша и приносит пользу.

Управляемая поведением разработка

Управляемая поведением разработка (Behavior-Driven Development, BDD) основывается на TDD и предоставляет разработчикам и другим задействованным специалистам, не имеющим отношения к программированию, общий язык для описания корректного поведения приложения или его отдельного модуля. При этом в качестве такого общего языка по сути выступает наш обычный повседневный язык, на котором мы с вами разговариваем и общаемся друг с другом.

Например, в рамках BDD вместо написания теста `testEmptyCart` вам нужно предоставить описание, определяющее поведение тестируемого модуля вроде этого: «Корзина в интернет-магазине не должна открываться, если она пуста». Общий язык для написания тестов и формирования ожиданий помогает любому специалисту понимать, что тестируется, а также определить, какими должны быть тесты и что должно получиться в результате теста.

Понятно, что желательно бы тесты писать хоть и на обычном языке, но в какой-то единообразной форме. С этой целью подход BDD использует форму пользовательских историй гибкой разработки в качестве базового шаблона для написания тестов. Соответственно, в рамках BDD пользовательские истории могут быть непосредственно переведены в тесты.

Формулируются пользовательские истории в следующем виде:

Как [кто-то] я хочу [что-то] так, чтобы [результат]

Небольшой пример: *«Как пользователь Почты Yandex я хочу присоединить изображение к своей электронной почте так, чтобы его видели мои получатели»*. Данная пользовательская история может быть с легкостью переведена в разряд требований и тестов Почты Yandex.

Кроме того, BDD благотворно влияет как на внутрикомандные коммуникации, так и на взаимодействия разных команд, поскольку позволяет участникам наиболее отчетливо понимать, как ваша система должна работать.

Какой подход лучше?

В рамках данной книги мы не ставим своей целью защищать или критиковать какую-либо из методик или методологий разработки. Водопадная, спиральная, гибкая и другие методологии хороши сами по себе, но ни одна из них не приводит к идеальному коду вообще и идеальному JavaScript-коду в частности. Соответственно, ни разработка через тестирование, ни управляемая поведением разработка тоже не гарантируют получения совершенного JavaScript-кода.

Как же получить хороший код фронтенда на JavaScript? На протяжении всей этой книги мы будем стараться ответить на этот вопрос и дать конкретные рекомендации/решения. Но для начала зададим вектор.

Пишите четкий, слабо связанный и хорошо прокомментированный код — это единственный верный путь к написанию хорошего JavaScript-кода фронтенда.

При этом использование любой из методик (будь то Test-Driven Development, Behavior-Driven Development или какой-либо еще «...-Driven» методики) хоть и не является панацеей, но настоятельно рекомендуется, поскольку, скажем, в случае с TDD тестирование при написании кода помогает избежать многих ошибок, а в случае с BDD при написании кода важно помнить, что вы не находитесь в вакууме, а реализуете вполне прагматичные пользовательские задачи.

Помните, что любой профессионально написанный код может и будет использоваться, поддерживаться и отлаживаться не только вами, но и другими людьми. Если это возможно, значит код хорош, если же с вашим кодом можете работать только вы – значит что-то вы делаете не так в этой жизни.

1.2. Код для людей

Код, который мы пишем, по большому счету предназначен не для компьютеров, а для людей.

Написание программного обеспечения – практический бизнес. Компьютерам нужны лишь биты. JavaScript, C++, Java, Perl, Lisp или что-либо еще – все это компилируется в чрезвычайно ограниченную систему команд центрального процессора. Процессор не знает, какой код он выполняет – «скомпилированный» или «интерпретируемый». Процессор не интересуют ни комментарии, ни точки с запятыми, ни пробелы. Процессор находится в блаженном неведении о структуре, синтаксисе или семантике любого из бесчис-

ленных языков программирования, используемых людьми. Программа на JavaScript очень похожа на программу на C++, которая в свою очередь с точки зрения процессора ничем не отличается от программы на Perl.

На самом низком уровне успех или отказ наших программ тесно переплетен с машинным кодом, который выполняет центральный процессор. Но мы редко его видим (если вообще видим). Мы видим лишь исходный код. И это, в общем-то, правильно.

1.2.1. Несколько вопросов «Почему?»

В общем случае программное обеспечение начинается с намерения, а разработка программного обеспечения (как и любая деятельность) по сути является комплексом ответов на вопросы. Что вы пытаетесь сделать? Что должна сделать та или иная часть кода? Зачем вы пишете код? Это очень важные вопросы, которые вы должны задавать себе каждый день. Вы преобразуете это начальное намерение (все ваши «что» и «почему») в фактический код: «как».

Первыми шагами к получению части «как» вашей работы является выяснение, что вы пытаетесь сделать и почему. Книги по языкам программирования помогают с самым низким уровнем «как». Тогда как книги, подобные этой, помогают справиться с высоким уровнем «как»: «как» написать совершенный код, «как» тестировать код и т.п. Но перед тем, как получить «как», важно понять «что» и «почему». Так почему же мы хотим написать хороший код фронтенда? И вообще что такое хороший код?

Почему хороший код — это код тестируемый?

«Написание программного обеспечения — самая сложная вещь, которую делают люди», — говорит Дуглас Крокфорд[2]. Чрезвычайно важно, чтобы программное обеспечение было максимально друже-

2 Дуглас Крокфорд – в американский программист и предприниматель, который известен своим постоянным участием в развитии языка JavaScript, в частности тем, что популяризировал формат данных JSON (JavaScript Object Notation), развивал и развивает различные, связанные с JavaScript, инструменты, такие как JSLint и JSMin. В настоящее время является главным JavaScript-архитектором в PayPal. Известен своими выступлениями на различных JavaScript/веб-конференциях, а также своей статьей «The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language» ((рус.) «Самый неправильно понятый язык программирования в мире стал самым популярным в мире языком программирования») о языке JavaScript.

ственным для человека. Максимально дружественным как для тех, кто им пользуется (пользователей), так и для тех, кто над ним работает (разработчиков) и поддерживает.

Хороший код – это код, который помимо грамотной реализации заложенной в него функциональности проще тестировать, проще обслуживать, он понятен и с ним проще работать.

Тестирование кода — обязательное действие. Тестировать можете вы лично, тестировать может другой программист или даже конечный пользователь, использующий ваше приложение. Ничто не совершенно в самом начале, а зачастую и вообще первый блин получается комом. Более того, ничто не идеально и в будущем — нет предела совершенству. Но есть определенные ориентиры и показатели, к которым нужно стремиться. Рассмотрим одну из простейших программ на JavaScript:

```
x = x + 1;
```

Даже в ней потенциально заложена куча проблем! А что, если `x` — строка? Что, если `x` — бесконечность? Что, если `x` — объект? И это только в одной строке простейшего кода. А поскольку наши программы становятся более сложными, мы можем только надеяться протестировать и предугадать наиболее вероятные проблемы. Исчерпывающее тестирование невозможно. Но нужно выполнить хотя бы самое основное тестирование и надеяться, что его будет достаточно.

Почему хороший код фронтенда – это код удобный в сопровождении?

Программисты не всегда пишут код с нуля. Очень часто нам приходится отлаживать и поддерживать чей-то код. При этом вам достается код, автор которого либо не доступен, либо сам уже не помнит, что он там делал, как и вы с трудом вспомните код, написанный год назад в своей предыдущей компании. Пока вы поддерживаете чей-то код, кто-то поддерживает ваш.

Код тем идеальнее, чем проще его поддерживать.

В ходе работы с унаследованным кодом, одной из основных проблем является то, что у разработчика нет ясного и полного понимания того, на что будут влиять внесенные им изменения. В итоге ему нужно постоянно проводить регрессионное тестирование[3], чтобы выяснить степень влияния, казалось бы, небольших изменений.

Когда же разработчик знает, как работает код и понимает, что произойдет после того, как он внесет в него свои изменения, — это и есть удобный в сопровождении код. Это такой код, которого вы не боитесь. Это код, который вы с легкостью можете совместно использовать с вашей командой. Это код, который не нужно переписывать заново, чтобы понять его.

По мере роста размера вашего приложения сокращается число людей, которые полностью ориентируются в вашем коде. И вот тогда начинают появляться индивидуумы, которые будут искренне удивляться тому, что на вид безвредное изменение в одном месте повлияло на функциональность в другом месте или вообще все поломало. Чтобы избежать или, по крайней мере, сократить количество таких моментов, ваш хороший код должен быть понятным. И тут мы переходим к следующему «Почему».

Почему хороший код фронтенда – это код понятный?

Третья особенность качественного программного обеспечения — понятный исходный код. Сколько времени нужно, чтобы понять, что делает тот или иной участок кода? Вы можете сказать, что на этот вопрос мы уже ответили в своем первом «Почему?», когда говорили о тестируемости: тестируемость и тестовое покрытие позволяет вам понять, что делает код. Отчасти это, конечно же, верно. С другой стороны, зачастую бывает важно понять не только, что делает тот или иной фрагмент кода, но и «как» он это пытается сделать. Если вы не можете понять чужой код или код, написанный вами, скажем, полгода назад, у вас серьезная проблема..

Еще раз повторю свою мантру: «код пишется для людей, которые должны понять, обслужить и протестировать его». Вы пишете код

3 Ошибки — когда после внесения изменений в программу перестает работать то, что должно было продолжать работать, — называют регрессионными ошибками.

для других людей, следовательно, ваши коллеги должны понимать его. Если код понятен, вы облегчите работу и себе, и вашим коллегам. Начните с этого прочного фундамента, и вы станете намного более продуктивным. Продуктивные люди — счастливые люди, а счастливые люди — продуктивны. Вот и получился такой добродетельный круг.

1.2.2. Несколько вопросов «Как?»

Как сделать код тестируемым? Связность кода

Написать хороший, тестируемый код фронтенда проще, когда вы начинаете с чистого листа. Некоторые методики, такие как TDD и BDD, могут привести к идеальному коду, но далеко не всегда. Дело в том, что наличие самих тестов автоматически не делает код идеальным. Даже если у вас будет большое число тестов, вы все еще можете написать очень плохой код. Однако, когда вы в самом начале думаете о тестировании, ваш код будет более совершенен, чем тот, который написан с нуля, но без всякой мысли о тестировании.

Тесты и код — это как курица и яйцо. Каждый из них может быть первым. Не пишите много тестов без кода и не пишите много кода без тестов. Вместо этого напишите немного кода и протестируйте его. Или же сначала напишите тесты, а затем — по ним — напишите код.

Во время написания кода и тестов старайтесь писать небольшие, изолируемые участки кода с минимальными зависимостями и минимальной сложностью. Подобное мышление — и есть суть этой книги.

Как сделать код удобным в сопровождении?

Способ достижения удобного в сопровождении кода почти такой же, как и способ достижения тестируемого кода: небольшие фрагменты кода, простой и изолируемый код.

Все просто: чем меньше строк кода, тем меньше ошибок; чем код проще, тем его проще сопровождать. Когда код изолируемый, внесенные в него изменения будут влиять на как можно меньшее количество других частей кода. В этой книге мы рассмотрим несколько методов, направленных на минимизацию размеров отдельных фрагментов кода и на снижение связности между ними.

Как сделать код понятным?

Как это ни удивительно, но написание понятного кода опять использует те же принципы, о которых мы говорили в предыдущих «Как'ах»: простота и низкая связанность – простой код проще и быстрее понять (написание кода подобно написанию глав романа: несколько небольших глав проще понять, чем несколько больших), тестируемость – наличие тестов вместе с кодом дает понимание того, что делает код. Плюс к этому добавим комментарии, которые значительно помогают в наглядности.

> **Немного подробностей как в отдельных комментариях (блоки с комментариями перед методами), так и непосредственно в коде (описывающие имена переменных) + непротиворечивый стиль кодирования улучшают восприятие кода.**

Ваши коллеги — не идиоты, но подсказки им все-таки лучше давать. Дайте им немного комментариев, которые помогут понять ваш код, и он не будет выброшен или переписан.

1.3. Что еще обеспечивает идеальность кода?

Ваша работа, зачастую, не ограничивается лишь написанием кода, Вы еще должны и протестировать его! Именно это вкупе с последующей отладкой приблизит его к совершенству. При этом высока вероятность того, что и то и другое вам придется делать самим, независимо от проекта. Написание тестов вы будете делать сами из соображений того, что держа в голове свои тесты вы лучше напишете код, да и ошибки обнаружите эффективнее, а дебажить вы будете, скорее всего, сами просто потому, что ни один разработчик не захочет отлаживать чужой код.

В заключение данной главы скажем пару слов и о тестировании, и об отладке.

Тестирование

Первая линия обороны разработчика – это *модульное тестирование*. **Модульное тестирование заставляет вас понять свой же код. Кроме того, модульное тестирование помогает вам документировать и отлаживать свой код.**

Интеграционное тестирование **поможет убедиться, что все в вашем приложении взаимодействует, как запланировано, особенно это касается клиентской части JavaScript, которая может работать в различных браузерах и на разных платформах (настольные компьютеры, планшеты и телефоны).**

Наконец, *тестирование производительности и нагрузочное тестирование* **призваны справиться с запланированным уровнем нагрузки.**

Каждая ступенька в лестнице, состоящей из всевозможных тестов, позволяет протестировать код на разных уровнях абстракции. Каждый тест находит ошибки в различных сценариях использования приложения. Полное тестирование требует тестирования на всех уровнях абстракции. Тем не менее, необходимо отдавать себе отчет, что даже после всех этих тестов у вас все равно будут ошибки, которые вы не обнаружили. Нет никакого волшебного способа гарантированно найти и исправить сразу все ошибки.

Отладка

Независимо от проведенных вами числа тестов, отладка – неотъемлемая часть жизни разработчиков программного обеспечения. К счастью, у JavaScript есть отличные инструменты, существенно упрощающие процесс отладки и облегчающие вашу жизнь. Но, даже несмотря на это, вполне может быть, что отладка кода займет больше времени, чем его написание. Далее, в соответствующей главе мы более подробно остановимся на отладке.

Резюме

Ни одна из существующих методологий разработки, будь то водопадная разработка, гибкая, TDD/BDD и проч. и проч., не позволяет получить хороший JavaScript-код фронтенда. Хороший JavaScript-код фронтенда – это код, который состоит из небольших, слабо связанных фрагментов с хорошими комментариями. То, как именно вы получите такой код, — дело ваше. Данная книга поможет вам понять, как вообще этого можно достичь.

Зачем вам все это? Написание хорошего JavaScript-кода облегчит вашу жизнь и жизнь всех тех, кому придется модифицировать и поддерживать ваш код. Сделает вас более эффективным, а ваш проект более успешным. Хороший JavaScript-код — это ваша дверь к здравомыслию в разработке на JavaScript и профессионализму разработчика вообще.

Не забывайте, что вы пишете код для людей, а не для компилятора. Эти люди будут поддерживать ваш код!

Глава 2.

Сложность кода

Сложность — это плохо. Простота — это хорошо. С некоторой точностью мы можем измерить сложность кода, используя статический анализ.

Статический анализ кода (англ. static code analysis) — это анализ программного обеспечения, производимый без реального выполнения исследуемых программ (в отличие от динамического анализа).

При этом нам на помощь приходят такие показатели и инструменты, как статический анализатор кода JSLint, цикломатическая сложность, число строк кода, разветвление на входе и выходе. В то же время не стоит забывать, что зачастую ничто так точно не измерит сложность вашего кода, как обычный показ его вашим же коллегам.

Порой нам приходится тратить примерно 50% своего времени на тестирование и сопровождение своего же кода, что по сути дела больше, чем занимает само его написание. А если с вашим кодом станет работать кто-то еще? Сложность — это яд каждого проекта. Вы должны стремиться сделать код максимально простым, как для себя, так и для других.

Почему код получается сложным? По многим причинам, от серьезных вроде «у нас запутанный алгоритм» до довольно приземленных вроде «этот JavaScript-код просто отвратительный, его написал новичок».

Удобный в сопровождении JavaScript-код должен быть четким, непротиворечивым, и основанным на стандартах, быть слабо связанным и лаконичным.

Как сказал Архимед, «Дайте мне точку опоры, и я переверну Землю», так вот, анализ сложности – эта та точка опоры, которая поможет сделать ваш код идеальным. В любом случае, некоторая сложность неизбежна, однако в большинстве ситуаций чрезмерной сложности можно избежать. Способность распознания сложных частей вашего приложения и понимание, почему они сложные, позволит вам понизить общую сложность проекта. Все как обычно: осознание проблемы — первый шаг к ее решению.

Важно отметить тот факт, что сложность может быть заложена в алгоритмах, которые реализует ваше приложение. Более того, зачастую эти «хитрые» алгоритмы являются тем, что делает ваш код уникальным и полезным, а ваше приложение выделяет среди конкурентов. Если бы базовые алгоритмы не были сложны, кто-то, вероятно, уже бы создал ваше приложение до вас.

2.1. Размер кода как характеристика сложности

Размер кода

С ростом кода увеличивается его сложность и уменьшается число людей, понимающих всю систему. С ростом числа модулей интеграционное тестирование становится все более и более трудным, также растет число перестановок взаимодействующих модулей. Не удивительно, что большой размер кода — индикатор номер один потенциальных ошибок:

Чем больше размер кода, тем выше шансы наличия ошибок в нем. Это утверждение относится как к программе в целом, так и к отдельным ее фрагментам. Чем меньше строк кода в завершенном фрагменте программы, тем он понятнее.

При этом параметр «количество строк кода» не является единственным показателем объема кода. Так, общее число кода, необходимого для реализации вашего проекта, может не изменяться, но зато может изменяться число операторов в методах и проч.

Большие методы сложно тестировать и сопровождать, поэтому старайтесь делать их более компактными. При написании кода важно думать о том, как вы будете его тестировать. В идеальном мире у функций нет побочных эффектов и возвращаемое ими значение (значения) целиком и полностью зависит только от переданных им параметров, но реальный мир функций далек от идеала, зависит от многих факторов помимо переданных параметров и зачастую образует большое количество проблемных исключительных ситуаций. Поэтому очень важно бывает локализовать проблему, что гораздо проще сделать в рамках небольших функций, чем блуждать по одной «здоровой» функции со многими аргументами и запутанной многовариантной функциональностью.

Минимизация размера кода

> **Один из методов минимизации кода функций — разделение их на команды и запросы[1]. Команды — это функции, которые** *делают что-то, что приводит к изменениям в состоянии какого-либо объекта*. **Запросы — это функции, которые просто** *возвращают что-то*.

В этом контексте команды используются как методы **set** (устанавливают что-то), а запросы – как методы **get** (возвращают что-то). Команды можно тестировать с использованием объектов-имитаций (`mocks`-объектов), а запросы – с использованием объектов-заглушек (`stubs`) (см. гл. 4).

Подобное разделение на команды и запросы, кстати говоря, помимо уменьшения размера кода самих функций, вместе с улучшением тестируемости, может обеспечить и большую масштабируемость за счет архитектурного отделения «блоков чтения» от «блоков записи». Масштабируемость такого архитектурного решения существенно лучше.

1 Этот принцип введен еще в прошлом веке Бертраном Мейером и носит название CQRS.

В качестве примера рассмотрим разделение команд и запросов в контексте применения Node.js[2]:

```
function configure(values) {
        var fs = require('fs')
                , config = { docRoot: '/somewhere' }
                , key
                , stat
        ;
for (key in values) {
        config[key] = values[key];
}
try {
        stat = fs.statSync(config.docRoot);
        if (!stat.isDirectory()) {
                throw new Error('Не допустимо');
        }
        } catch(e) {
                console.log("** " + config.docRoot +c
                " не существует или это не каталог!!! **");
                return;
        }
// ... проверяем другие значения ...
return config;
}
```

Давайте рассмотрим тестовый сценарий для этой функции. В главе 4 мы обсудим синтаксис этих тестов более подробно, а сейчас мы просто посмотрим на него:

```
describe("настраиваем тесты", function() {
    it("не определено, если docRoot не существует", function() {
            expect(configure({ docRoot: '/xxx' })).toBeUndefined();
    });
    it("определено, если docRoot существует", function() {
    expect(configure({ docRoot: '/tmp' })).not.toBeUndefined();
    });
    it("добавляем значения в хэш config", function() {
    var config = configure({ docRoot: '/tmp', zany: 'crazy' });
            expect(config).not.toBeUndefined();
```

2 Node.js – это серверная платформа (каркас, фреймворк) на базе JavaScript, предназначенная для создания масштабируемых распределённых сетевых приложений, таких как веб-сервер. В отличие от большинства JavaScript-программ, этот фреймворк исполняется не в браузере клиента, а на стороне сервера.

```
            expect(config.zany).toEqual('crazy');
            expect(config.docRoot).toEqual('/tmp');
    });
    it("проверка value1 успешно...", function() {
    });
    it("проверка value1 не успешна...", function() {
    });
    // ... ожидается много других проверок...
    });
```

Наша изначальная функция делает слишком много. После установки значений конфигурации по умолчанию она проверяет корректность этих значений; при этом фактически при тесте проверяется более пяти значений. Метод у нас большой, а каждая проверка полностью независима от предыдущих проверок, то есть вся логика проверки допустимости для каждого значения впихнута в эту единственную функцию. Проверить какое-то значение изолированно невозможно. Таким образом, тестирование инициирует множество проверок всех возможных значений при проверке каждого конфигурационного значения.

Точно так же эта функция требует множества модульных тестов со всеми проверками в рамках модульного теста основного функционала самой функции. Это плохо. Особенно на фоне того, что, скорее всего, спустя некоторое время будет добавлено еще больше конфигурационных значений, а значит, данная проверка станет еще более уродливой. Также, погружение ошибок в блоки `try/catch` простонапросто выводит эти блоки из строя.

Наконец, сбивает с толку возвращаемое значение: или оно не определено, если какое-то значение не было проверено, или это весь допустимый хэш. И надо не забывать о побочных эффектах операторов `console.log` (но об этом попозже).

Давайте посмотрим, как описанные проблемы могут быть разрешены за счет разделения функции на несколько частей. Вот один из вариантов:

```
function configure(values) {
    var config = { docRoot: '/somewhere' }
        , key
    ;
    for (key in values) {
```

```
                config[key] = values[key];
        }
        return config;
}
function validateDocRoot(config) {
        var fs = require('fs')
                , stat
        ;
        stat = fs.statSync(config.docRoot);
        if (!stat.isDirectory()) {
                throw new Error(Не допустимо');
        }
}
function validateSomethingElse(config) { ... }
```

Здесь мы разделили: отдельно установка значений (**запрос**, возвращаемое значение) и отдельно: описание функций проверки (**команда**, обработчик ошибок и проч.). Разделение исходной большой функции на две меньших функции делает тестирование нашего модуля более сфокусированным и более гибким.

Теперь мы можем написать отдельный, изолированный тест для каждой функции вместо написания одного огромного теста:

```
describe("validate value1", function() {
        it("accepts the correct value", function() {
                // некоторые исключения
        });
        it("rejects the incorrect value", function() {
                // некоторые исключения
        });
});
```

Мы делаем заметный шаг на пути уменьшения размера кода в отдельно взятых функциях, а также большой шаг к хорошей тестируемости, поскольку функции проверки допустимости того или иного значения могут теперь проводиться обособленно, не будучи в составе более общих конфигурационных (configure) тестов. Все это приближает наш код к идеалу. Однако отделение методов-сеттеров (установки значений) от методов проверки не всегда является самоочевидным. В нашем случае выполненное нами разделение команд и запросов может быть не очень хорошим решением, если мы за-

хотим обезопасить себя от побочных эффектов установки значений без их одновременной проверки. Поэтому, хотя разделение функций – отличный прием для повышения тестируемости кода и снижения его сложности, нам нужно убедиться, что оно будет реализовано корректно.

Следующая итерация будет примерно такой:

```
function configure(values) {
        var config = { docRoot: '/somewhere' };
        for (var key in values) {
                config[key] = values[key];
        }
        validateDocRoot(config);
        validateSomethingElse(config);
        ...
        return config;
}
```

Данная новая конфигурационная функция или возвращает допустимый объект `config`, или порождает ошибку. Все функции проверки (`validateDocRoot, validateSomethingElse,...`) могут быть протестированы отдельно из самой функции `configure`.

Последний штрих – мы должны соединить каждый ключ объекта `config` с его функцией проверки и хранить весь хэш в одном централизованном месте:

```
var fields {
        docRoot: { validator: validateDocRoot, default: '/somewhere' }
        , somethingElse: { validator: validateSomethingElse }
};
function configure(values) {
        for (var key in fields) {
                if (typeof values[key] !== 'undefined') {
                        fields[key].validator(values[key]);
                        config[key] = values[key];
                } else {
                        config[key] = fields[key].default;
                }
        }
        return config;
}
```

У нас получился отличный компромисс. Функции проверки теперь доступны отдельно для упрощения тестирования и снижения сложности кода, они будут вызываться при установке новых значений, а все данные хранятся в одном централизованном расположении.

Но, с другой стороны, у нас все еще есть проблема. Что помешает кому-то ввести следующее:

```
config.docRoot = '/does/not/exist';
```

Наша функция проверки не будет работать. Есть отличное решение: использовать новые методы `Object` из ECMAScript 5 и 6[3]. Они позволяют создавать свойства объекта со встроенными функциями проверки, методами get/set и многое другое, и к тому же они довольно хорошо тестируемые:

```
var obj = { realRoot : '/somewhere' };

Object.defineProperty(obj, 'docRoot',
        {
                enumerable: true
                , set: function(value) {
                        validateDocRoot(value); this.realRoot = value; }
        }
);
```

Теперь при следующем вводе:

```
config.docRoot = '/does/not/exist';
```

будет запущена функция `set`, которая вызовет функцию проверки, которая породит исключение в случае отсутствия указанного пути, поэтому проверка прервана не будет.

Но это странно. Вышеуказанный оператор присваивания мог бы теперь породить исключение и должен быть заключен в блок `try/catch`. Даже если вы избавитесь от вызова исключения, во что вы теперь установите `config.docRoot`, если значение не пройдет проверку? Независимо от установленного значения, результат будет неожиданным. А неожиданные результаты становятся проблемой.

3 ECMAScript — это встраиваемый расширяемый не имеющий средств ввода/вывода язык программирования, используемый в качестве основы для построения других скриптовых языков. Стандартизирован международной организацией ECMA. Язык JavaScript является расширением языка ECMAScript.

К тому же сбивает с толку странность внутренних и внешних имен `docRoot` и `realRoot`.

Лучшее решение – использовать закрытые (private) свойства с методами установки и получения значений. В этом случае свойства будут закрытыми, но все остальное будет открытым (public), в том числе функция проверки – для лучшей тестируемости:

```
var Obj = (function() {
        return function() {
                var docRoot = '/somewhere';
                this.validateDocRoot = function(val) {
                // логика проверки – вызвать исключение, если не OK
                };
                this.setDocRoot = function(val) {
                        this.validateDocRoot(val);
                        docRoot = val;
                };
                this.getDocRoot = function() {
                        return docRoot;
                };
        };
}());
```

Теперь доступ к свойству `docRoot` возможен только через ваш API, что приводит к проверке при записи. Используйте его примерно так:

```
var myObject = new Obj();
try {
        myObject.setDocRoot('/somewhere/else');
} catch(e) {
        // что-то не так с новым значением docRoot
        // старое значение docRoot все еще здесь
}
// все OK
console.log(myObject.getDocRoot());
```

Метод `set` теперь заключен в блок `try/catch`, что более ожидаемо, чем заключение в блок `try/catch` оператора присваивания.

Но осталась еще одна проблема, которую мы можем решить, — что насчет этого:

```
var myObject = new Obj();
myObject.docRoot = '/somewhere/wrong';
// и затем позже...
var dR = myObject.docRoot;
```

Конечно, ни один из методов в API не будет знать об этом ошибочно созданном пользователем поле `docRoot`. Что есть нехорошо. К счастью, это очень легко исправить:

```
var Obj = (function() {
        return function() {
                var docRoot = '/somewhere';
                this.validateDocRoot = function(val) {
                // логика проверки – исключение, если не OK
                };
                this.setDocRoot = function(val) {
                        this.validateDocRoot(val);
                        docRoot = val;
                };
                this.getDocRoot = function() {
                        return docRoot;
                };
                Object.preventExtensions(this)
        };
}());
```

Используя `Object.preventExensions`, мы породим исключение `TypeError`, если кто-то попытается добавить свойство в объект. В результате не будет никаких поддельных полей (свойств). Это очень удобно для всех ваших объектов, особенно если у вас есть непосредственный доступ к свойствам. Интерпретатор теперь будет перехватывать любые ошибочно добавленные свойства путем порождения `TypeError`.

Этот код также отлично инкапсулирует разделение команд и запросов с методами `set/get/validator`: все раздельны и каждый тестируется изолированно.

Функции проверки должны быть закрытыми, но для чего? Лучше сохранять их открытыми не только из соображений тестирования, но и чтобы производственный код мог проверить, является ли значение `docRoot` допустимым или нет без необходимости его явной установки на объекте.

Далее приведен чистый код тестового сценария – краткий, понятный:

```
describe("validate docRoot", function() {
        var config = new Obj();
        it("ошибка, если docRoot не существует", function() {
                expect(config.validateDocRoot.bind(config, '/xxx')).toThrow();
        });
        it("нет ошибки, если docRoot существует", function() {
                expect(config.validateDocRoot.bind(config, '/tmp')).not.toThrow();
        });
});
```

Разделение команд и запросов – не единственное средство оптимизации, а использование его не всегда возможно, но зачастую – это хорошая отправная точка в том, чтобы сделать ваш код идеальным. И хотя размером кода функций можно управлять и многими другими способами, разделение кода на команды и запросы очень хорошо соотносится с ключевым преимуществом, достоинством JavaScript: обработкой событий. Подробнее об этом мы поговорим в следующей главе.

2.2. Статический анализатор кода JSLint

Непосредственно для измерения и изменения сложности кода JSLint не предназначен, скорее он принуждает вас понимать, что делает ваш код. Именно это в свою очередь снижает сложность и гарантирует, что вы не используете сложные или подверженные ошибкам конструкции. Создание JSLint было вдохновлено наличием анализатора lint для языка C, и аналогично оригинальному "lint" для C, анализатор JSLint анализирует код на наличие плохого стиля, синтаксиса и семантики. Короче, он обнаруживает плохие участки вашего кода.

Рассмотрим простой пример:

```
function sum(a, b) {
        return
                a+b;
}
```

Давайте "скормим" его JSLint:

```
Error:
Problem at line 2 character 5: Missing 'use strict' statement.
return
Problem at line 2 character 11: Expected ';' and instead saw 'a'.
return
Problem at line 3 character 8: Unreachable 'a' after 'return'.
a+b;
Problem at line 3 character 8: Expected 'a' at column 5,
not column 8.
a+b;
Problem at line 3 character 9: Missing space between 'a' and '+'.
a+b;
Problem at line 3 character 10: Missing space between '+' and 'b'.
a+b;
Problem at line 3 character 10: Expected an assignment or
function call and instead saw an expression.
a+b;
```

Вау! Четыре строки простейшего кода сгенерировали целых семь ошибок JSLint! Что не так? Прежде всего, отсутствует директива `use strict`, но самая большая проблема – символ возврата каретки после `return`. Из-за вставки точки с запятой JavaScript вернет значение `undefined` из этой функции. JSLint зафиксировал эту ошибку и жалуется на другие пробелы в этой функции.

Пробел важен для удобочитаемости, но в нашем случае (см. код выше) он вызывает ошибку, которую очень трудно обнаружить. Используйте пробелы разумно, как требует JSLint.

Вот как выглядит правильный код:

```
function sum(a, b) {
        return a + b;
}
```

Давайте посмотрим еще на один фрагмент, безвредный на первый взгляд:

```
for (var i = 0; i < a.length; i++)
        a[i] = i*i;
```

На сей раз JSLint не сможет даже проанализировать весь блок, и только доберется сюда:

```
Error:
Problem at line 1 character 6: Move 'var' declarations to the top of the function.
for (var i = 0; i < a.length; i++)
        Problem at line 1 character 6: Stopping. (33% scanned).
```

Первая проблема – это объявление `var` в цикле `for`. Область действия переменной в JavaScript может быть или глобальной или локальной. Объявление переменной в цикле `for` не означает объявление переменной только для цикла `for`. Переменная будет доступна внутри всей функции, содержащей цикл `for`.

Используя вышеприведенное объявление, вы только что создали переменную с именем `i`, которая доступна везде до закрытия функции. Возникает неопределенность в понимании области действия переменной (и того, что именно этим хотел сказать разработчик таким объявлениям, и понимает ли он вообще, что делает), а также затрудняется контроль за переменными, действие которых распространяется на всю функцию, а объявления разбросаны по всему телу функции.

Написание кода в подобном стиле сбивает с толку не только вас самих, но и другого программиста, который будет поддерживать этот код, что еще хуже. Поэтому JSLint просит вас переместить все объявления переменных в начало функции.

Примечание.

Хотя это и не очень критично, вы должны избавиться от привычки объявления переменных где угодно, кроме как в начале функции. Переменные в JavaScript действуют и объявляются (описываются) по всей функции, и в этом разница между JavaScript и другими языками программирования. Но пользоваться этим надо обдуманно. Просто переместите объявление переменной в начало функции, где это целесообразно.

Вернемся к нашему примеру, допустим, что мы переместили объявление переменной вверх, что нам JSLint скажет теперь? А вот что:

```
Error:
        Problem at line 2 character 28: Unexpected '++'.
```

```
for (i = 0; i < a.length; i++)
        Problem at line 3 character 5: Expected exactly one space between
')' and 'a'.
a[i] = i*i;
        Problem at line 3 character 5: Expected '{' and instead saw 'a'.
a[i] = i*i;
        Problem at line 3 character 13: Missing space between 'i' and '*'.
a[i] = i*i;
        Problem at line 3 character 14: Missing space between '*' and 'i'.
a[i] = i*i;
```

JSLint'у не понравился оператор ++. Считается, что префиксные/постфиксные операторы ++ и -- могут запутать разработчика. И хотя это не самые критичные проблемы и JSLint здесь немного категоричен, но эти операторы — пережиток прошлого от C/C++ и в JavaScript их рекомендуется не использовать. В случаях, когда такие операторы потенциально могут запутать программиста, нужно отказаться от их использования.

JSLint также хочет заключить в фигурные скобки тело цикла. Вероятнее всего, тело цикла не будет состоять из одной строки, а будет расширяться в будущем, а отсутствие скобок будет кого-то сбивать с толку — причем не только программистов, но и некоторые инструменты статического анализа (инструменты статического анализа могут запутаться без фигурных скобок вокруг циклов). Сделайте полезное дело для всех, в том числе и для себя, используйте фигурные скобки. Два байта «расходов», необходимых на это полезное дело, окупятся читабельностью кода:

```
for (i = 0; i < a.length; i = i + 1) {
        a[i] = i * i;
}
```

То, что особенно опасно в нашем исходном коде, – это то, что формально в нем нет никаких ошибок! Он будет скомпилирован, запущен и сделает то, что от него и ожидается. Но в будущем скрытый беспорядок и плохая читабельность исходного кода станут обузой. Особенно когда размер кода возрастет до нескольких сотен тысяч строк и код станет поддерживается другими программистами (или даже самим автором, спустя шесть месяцев, когда он забудет все, что имел в виду при написании этого кода).

В данной книге мы не будем подробно рассматривать все возможные настройки и варианты использования JSLInt. За тем и другим рекомендуем отправиться на официальный веб-сайт JSLInt (http://jslint.com/), а также рекомендуем обратиться к книге **JavaScript: The Good Parts** (автор Крокфорд)[4], в которой приводятся «плохие» и «хорошие» синтаксические конструкции JavaScript.

Для кого-то анализатор JSLint может показаться слишком «строгим» (выдающим слишком много некритичных замечаний), в таких случаях рекомендуем обратить свое внимание на JSHint (http://www.jshint.com/), который является более гибкой и более «прощающей» производной JSLint.

Обратите внимание, что JSLint может также считать параметры конфигурации в комментариях в начале ваших JavaScript-файлов, поэтому не стесняйтесь настраивать их. Что касается самих настроек, то я рекомендую использовать стандартный набор параметров конфигурации JSLint везде при анализе вашего приложения.

В качестве небольшого отступления отметим, что существенная проблема при использовании JSLint состоит в том, что он не допускает использование цикла `for`, поскольку этот цикл он не считает идиоматическим для JavaScript (проще говоря, конструкция `for` не является родной для JavaScript). Однако, используя инструменты, предоставляемые языком, вы можете решить все проблемы, возникающие с исходным циклом в JSLint:

```
a.forEach(function (val, index) {
        a[index] = index * index;
});
```

Использование метода `forEach` гарантирует, что значение и индекс массива будут должным образом ограничены просто вызовом функции `function`. Еще важен тот факт, что эта конструкция показывает специалистам, которые будут обслуживать ваш код в будущем, что вы знаете, что вы делаете, и используете идиоматический JavaScript.

Однако, хотя это немедленно исправляет все проблемы, которые у нас есть с исходным кодом, к сожалению, на момент написания этих строк метод `forEach` не поддерживается во всех браузерах. Напри-

4 На русском языке издано издательством «Питер» в 2012 г. под названием «JavaScript: сильные стороны». ISBN 978-5-459-01263-7.

мер, IE 8 и более ранние версии этого браузера не имеют собственной поддержки `forEach`. Но не стоит беспокоиться. Ведь его достаточно просто добавить. При этом не нужно пытаться написать метод самостоятельно, просто используйте стандартную реализацию Mozilla, доступную по адресу

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/forEach

Очень важно отметить, что предложенная реализация этой `callback-функции` получает три аргумента: значение, текущий индекс и сам объект, а не только элемент массива. Вы можете легко написать базовую версию `forEach`, которая будет занимать гораздо меньше строк, но не нужно этого делать! Используйте основанную на стандартах версию. Для надежности вы можете просто добавить этот фрагмент кода в начало вашего JavaScript-кода, чтобы гарантировать, что `forEach` будет доступен в каждом контексте.

В то же время, если вы уверены, что ваш код не будет использоваться в окружениях без родной поддержки `forEach` или же вы пишете JavaScript для серверной стороны, вы можете не добавлять этот фрагмент кода. Добавление реализации этого метода, конечно же, увеличит размер кода, который должен будет каждый раз загружаться и анализироваться. Однако после минимизации он сжимается всего до 467 байтов и будет полностью выполнен всего лишь один раз на том же IE 8 или более ранних версиях.

В результате мы получаем хорошее компромиссное решение, в котором у стандартного метода `forEach` есть переменная `index`, область действия которой ограничена самой функцией (`callback-функцей`), и у нас гарантированно не будет конфликтов переменных в случае описания такой же переменной в вызывающем методе. Также нам не нужно иметь дело с пост-условным оператором `++` и фигурными скобками.

Используя «хорошие» функции JavaScript и программируя идиоматически, вы создаете чистый, понятный и тестируемый код, а значит, делаете его «идеальным».

2.3. Цикломатическая сложность

Цикломатическая сложность — это мера числа линейно независимых путей в вашем исходном коде. Цикломатическая сложность равна увеличенному на единицу цикломатическому числу графа программы. Где граф программы – это как раз граф путей реализации вашей программы, а цикломатическое число – минимальное число ребер, которые надо удалить, чтобы граф стал ациклическим (то есть без циклов).

С точки зрения тестирования цикломатическая сложность — это минимальное число модульных тестов, необходимых для тестового покрытия всего вашего кода. Рассмотрим следующий пример:

```
function sum(a, b) {
        if (typeof(a) !== typeof(b)) {
                throw new Error("Не могу вычислить сумму разных типов!");
        } else {
                return a + b;
        }
}
```

Цикломатическая сложность этого метода равна 2. Это означает, что вам нужно написать два модульных теста, чтобы покрыть все 100% кода.

Примечание.

В приведенном выше коде мы используем оператор `!==` для проверки типов аргументов с помощью `typeof`. Хотя это не строго необходимо, использование операторов `!==` и `===` является хорошей привычкой. Использование строгого равенства (`===` и `!==`) поможет быстро обнаруживать ошибки в вашем коде.

Вычислить цикломатическую сложность вашего кода можно, например, утилитой командной строки под названием **jsmeter**

(https://code.google.com/p/jsmeter/). Однако определить, какое значение цикломатической сложности является оптимальным в том или ином случае — не так просто. Например, в статье «Измерение сложности» (**http://www.literateprogramming.com/mccabe.pdf**) ее автор, Томас Дж. Маккейб, сообщает, что ни у одного из методов не должно быть цикломатической сложности больше 10. В то же время исследование, с которым вы можете ознакомиться по ссылке **http://www.enerjy.com/blog/?p=198**, говорит, что цикломатическая сложность и вероятность ошибок не будут коррелировать до тех пор, пока цикломатическая сложность не достигнет значения 25 или выше.

Десять — это не магическое число, а, скорее, обусловленное здравым смыслом значение, поэтому, несмотря на то, что корреляция между кодом и ошибками начнется, только когда цикломатическая сложность достигнет 25 и выше, из соображений здравого смысла и пригодности для обслуживания кода в дальнейшем, желательно сохранить это число ниже 10.

Код с цикломатической сложностью больше 25 является очень сложным. Независимо от текущего числа ошибок в вашем коде, редактирование метода с высокой сложностью почти всегда вызовет ошибку.

Таблица 2.1 показывает, как Aivosto.com измеряет вероятность «плохой правки» – «bad fix» (изменения в коде, приведшие к ошибке) – при увеличении цикломатической сложности.

Таблица 2.1. Цикломатическая сложность и вероятность возникновения ошибок при правке кода

Цикломатическая сложность	Вероятность «плохой правки» («bad fix»)
1-10	5%
20-30	20%
>50	40%
Почти 100	60%

Из этой таблицы следует интересный факт. При правке относительно простого кода есть 5%-ный шанс, что вы допустите новую ошибку. Это существенно! Возможность допустить ошибку вырастает до 20% при росте цикломатической сложности свыше 20. Я никогда не

видел функции с цикломатической сложностью выше 50, не говоря уже о 100. Если вы замечаете, что цикломатическая сложность приближается к любому из тех чисел, вы должны бить тревогу! Как отмечает Маккейб в вышеупомянутой статье, функции со сложностью выше 16 являются также и менее надежными.

Как уже было сказано, чтение чьего-либо кода зачастую является очень важным индикатором качества кода и его правильности. Читатель должен понимать, что делают все ветки кода, только взглянув на него. Многие ученые проводили исследования кратковременной памяти. Самым известным из этих постулатов является закон оперативной памяти человека «7 плюс/минус 2 элемента» (также известный как закон Миллера), гласящий о том, что человек единовременно в фокусе своего внимания может держать 7+-2 элемента. Однако, для профессионального программиста, читающего код, значение цикломатической сложности может быть выше этого значения на короткие промежутки времени.

Высокая цикломатическая сложность происходит обычно из-за большого количества операторов `if/then/else` **(или операторов** `switch`**, но вы же их не используете, я надеюсь, не так ли?). Самый простой рефакторинг позволяет разбить код на меньшие методы или же использовать так называемую таблицу поиска**[5].

Рассмотрим небольшой пример:

```
function doSomething(a) {
        if (a === 'x') {
                doX();
        } else if (a === 'y') {
                doY();
        } else {
                doZ();
        }
}
```

5 Таблица поиска (англ. lookup table) — это структура данных, обычно массив или ассоциативный массив, используемая с целью заменить вычисления на операцию простого поиска. Увеличение скорости может быть значительным, так как получить данные из памяти зачастую быстрее, чем выполнить трудоёмкие вычисления.

Данный код можно переписать следующим образом с использованием таблицы поиска:

```
function doSomething(a) {
        var lookup = { x: doX, y: doY }, def = doZ;
        lookup[a] ? lookup[a]() : def();
}
```

Заметьте, что, проведя рефакторинг условного выражения в таблицу поиска, мы не сократили цикломатическую сложность (а значит, не сократили и число необходимых модульных тестов и не упростили сопровождение). Методы с высокой цикломатической сложностью лучше разбивать на несколько меньших методов. После чего у вас будет несколько функций с более низкой цикломатической сложностью, а такие функции значительно более удобны в сопровождении.

Добавление проверки цикломатической сложности в ваш процесс сборки осуществляется довольно просто. Просто посмотрите на ваш код и примите соответствующие меры. Для существующего ранее проекта разбейте сначала самые сложные методы и объекты, **но ни в коем случае не делайте это перед написанием модульных тестов, только после – сначала должны быть готовы модульные тесты!**

Правило номер один рефактринга — «не причини вреда», но вы не можете быть уверены, что следуете этому правилу, пока у вас не будет модульных тестов, чтобы проверить версии «до» и «после» изменений вашего кода.

Очень полезна для вычисления цикломатической сложности каждой вашей функции и каждого вашего метода утилита **jscheckstyle (https://github.com/nomiddlename/jscheckstyle)**. В главе 8 мы интегрируем ее с утилитой **Jenkins**, чтобы автоматически отмечать код, который может быть чрезмерно сложным и готовым к рефакторингу. Сейчас же рассмотрим утилиту саму по себе.

Следующая команда устанавливает пакет **jscheckstyle**:

```
% sudo npm install jscheckstyle -g
```

Теперь запустим эту утилиту для проверки JavaScript-файла:

```
findresult-lm:~ trostler$ jscheckstyle firefox.js
```

```
The "sys" module is now called "util". It should have a similar inter-
face.
jscheckstyle results - firefox.js
  Line        Function          Length      Args      Complex
    8      doSeleniumStuff        20          0           1
   26        Anonymous             1          0           1
   31       startCapture           17         1           1
   39        Anonymous             6          1           1
   40        Anonymous             4          1           1
   44        Anonymous             3          1           1
   49           doWS               40         2           2
   62        ws.onerror            4          1           1
   67        ws.onopen             4          1           1
   72       ws.onmessage           6          1           2
   79        ws.onclose            9          1           1
```

Утилита выводит список всех функций, присутствующих в файле, число строк в каждой из функций, число аргументов каждой функции и цикломатическую сложность каждой функции. Имейте в виду, что число строк подсчитывается с включением пустых строк и строк комментариев, поэтому данное значение как-то относительно в отличие от других показателей, значения которых являются абсолютными.

Обратите внимание, что в приведенном примере фактически есть только три высокоуровневых функции. Анонимная функция в строке 26 относится к функции `doSeleniumStuff`. Три анонимных функции также есть в функции `startCapture`, а функция `doWS` содержит четыре функции `ws.*`. Таким образом, этот инструмент, к сожалению, выявляет иерархию функций в файле довольно нечетко, поэтому будьте внимательны.

Напоследок отметим, что утилита **jscheckstyle** может также выводить информацию в форматах JSON и HTML, для чего при ее вызове нужно использовать параметры командной строки `-json` и `-html` соответственно.

2.4. Повторное использование кода

Лучший способ уменьшить размер кода заключается в уменьшении числа написанного вами кода.

Принцип, по которому предлагается достигать этого, заключается в том, что вы должны использовать сторонний (внешний или внутренний) код, готовый к использованию и обслуживаемый кем-то еще, что снимает с вас часть ответственности. Почти все используют сторонний (открытый или нет) код в своих программах. Джеффри Пулин – один из признанных экспертов в измерении и анализе повторного использования кода – отметил, что до 85% кода в программах не являются специфическими для приложения и только 15% кода программы делают ее уникальной. В его статье «Domain Analysis and Engineering: How Domain-Specific Frameworks Increase Software Reuse» (http://jeffreypoulin.info/Papers/JAPAN95/japan95.html) он пишет следующее:

Типичное приложение будет на 20% состоять из кода, который вообще никак не связан с областью применения приложения и носит общий характер, еще до 65% кода завязано на предметную область приложения, но допускает повторное использование кода как своего,

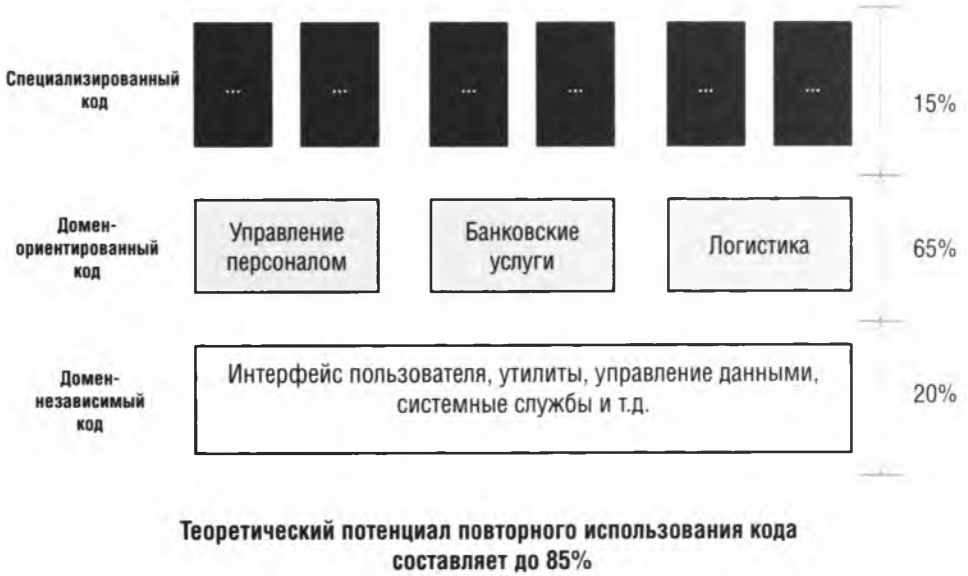


Рис. 2.1. Композиция типового приложения

так и чужого, созданного для этой предметной области. И только оставшаяся часть (порядка 15%) делает создаваемое приложение уникальным, реализует его уникальную функциональность, пишется конкретно под это приложение и мало применима где-либо еще.

Таким образом, можно выделить три типа программного кода в рамках совокупного кода приложений — специализированный, ориентированный на предметную область, и независимый от предметной области приложения — как показано в рисунке 2.1.

Для JavaScript кодом, независимым от контекста (от предметной области), являются например, используемые фреймворки: к примеру фреймворк YUI на стороне клиента или фреймворк Node.js на стороне сервера. Зависимый от предметной области приложения код представляет собой библиотеки, модули и проч., которые вы используете в своем приложении и которые были разработаны вами или третьими лицами безотносительно текущего приложения, но в целом ориентированы на данную предметную область. В любом случае при эффективной разработке вы этот код не пишете, а заимствуете/повторно используете. А пишете вы только специализированный код. При таком распределении трудовые затраты минимальны, а эффективность разработки достаточно высока.

Так, клиентский JavaScript-код предназначен для исполнения на многих платформах (под платформой мы понимаем «операционная система + браузер»), и попытка с нуля учесть все возможные комбинации в рамках разработки какого-либо веб-приложения является безумной и малореализуемой. Таким образом, использование какого-либо фреймворка, работающего на стороне пользователя, является насущной необходимостью почти в каждом случае. Иначе ваш проект зароется в проблемах, еще не успев как следует начаться. Благо решений, направленных на избавление от подобных проблем, сейчас довольно много. Такие фреймворки, как YUI, Closure и jQuery, содержат большие объемы универсального шаблонного кода, который позволяет разработчикам не отвлекаться на техническую рутину, а писать свой специализированный код, во многих случаях не заботясь о том, чтобы этот код мог выполняться на разнообразных платформах. За это отвечают фреймворки.

Далее, многие клиентские фрейворки содержат довольно обширный набор шаблонных компонент интерфейса, что позволяет очень

быстро и просто «лепить» интерфейсы ваших веб-приложений хотя бы в базовой функциональности. Причем интерфейсы сразу получаются относительно красивыми и непротиворечивыми. Кроме того, фреймворки зачастую предоставляют утилиты, плагины и дополнения, которые значительно уменьшают размер вашего «рукописного» кода. Так, на стороне сервера Node.js использует инструмент Node Package Manager (a.k.a. npm), чтобы использовать модули, загруженные в реестр npm (http://bit.ly/XUdvlF), благодаря чему функциональность Node.js может быть существенно расширена.

Далее, обработка событий, реализация которой является основной и чрезвычайно важной частью написания JavaScript-кода, также является потенциальным источником многих проблем, если попытаться обойтись без повторно используемого чужого кода. Так, ключевым для стороны клиента событием является обработка событий интерфейса пользователя браузера, а на стороне сервера фреймворк Node.js широко использует события обратных вызовов (callback), да и многие другие. Но в самой спецификации языка JavaScript нет средств для обработки событий. Поэтому разработку данных средств возложили на плечи разработчиков браузеров, которые активно конкурировали друг с другом. Не удивительно, что это привело к разделению и разбросу в решениях, пока не вмешался Консорциум World Wide Web (W3C), который попытался стандартизировать хотя бы сами клиентские события в JavaScript. Но обработчики событий по-прежнему остались отданными на откуп разработчикам.

Итак, необходимость обработать регистрацию события с одной стороны и нестыковки обработчиков событий в разных браузерах с другой стороны привели к тому, что исходная задача (регистрация событий) является довольно болезненной и ошибкоемкой. Самостоятельно ее решить довольно трудно, да и зачем заново изобретать колесо, когда использование какого-либо JavaScript-фреймворка быстро и легко решает подобные проблемы совместимости с браузерами. При этом вам предлагается готовое решение, пригодное не только к одному-двум браузерам (как это было бы, если бы вы писали подобный код самостоятельно), а «закрывает» вопрос совместимости практически со всеми современными браузерами, причем совершенно бесплатно.

Раз уж мы часто упоминаем Node.js, то отметим, что и в нем самом есть все возрастающее число сторонних модулей, которые вы можете использовать через npm. Полюбоваться на все это Node'шное многообразие и выбрать подходящий вам модуль для Node.js можно по адресу https://npmjs.org. Конечно, качество модулей варьируется, но даже если ничего не подойдет, то вы сможете посмотреть, как другие пытались решить подобную проблему, прежде чем начать писать свой код.

Помимо фреймворков в рамках программирования на языке JavaScript есть очень большое количество просто компонентов, доступных для использования. Таким образом, вам нужно стараться не пытаться все сделать самому с нуля, а где это возможно использовать готовые решения, созданные либо вами же чуть ранее, либо третьими лицами. Лучше сконцентрируйтесь на тех 15% кода, которые делают ваше приложение уникальным.

Более того, рекомендуется изначально писать код с расчетом на то, чтобы он мог использоваться вами еще где-то! В разделе «Разветвление на входе» мы обсудим, как можно обнаружить ваш код, который не используется повторно должным образом (наше обсуждение мы также продолжим в главе 8, когда рассмотрим инструмент **dupfind).**

Общее правило повторного использования кода заключается в следующем: если вы обнаружили, что дважды написали тот или иной код, его пора вынести в свою собственную функцию. Несмотря на то, что вы используете код всего дважды и вам лень создавать отдельную функцию для него, – все равно создайте ее – это принесет вам дивиденды в будущем. Практика показывает, что если код требуется в двух местах, то, скорее всего, со временем он понадобится в трех местах и т.д.

Повторное использование кода позволит сэкономить время не только разработки, но и время отладки/исправления кода.

2.5. Разветвление на выходе (Fan-Out)

Разветвление на выходе (Fan-Out) – это мера числа модулей или объектов, которые прямо или косвенно зависят от вашей функции.

Разветвление на выходе (как и разветвление на входе) было изучено в 1981 Салли Генри (Sallie Henry) и Деннисом Кэфурой (Dennis Kafura), которые написали о них в статье "Software Structure Metrics Based on Information Flow", опубликованной в журнале IEEE Transactions on Software Engineering. Они пришли к выводу, что от 50 до 75% стоимости программного обеспечения закладывается бизнесом (заказчиком ПО) в его обслуживание/поддержку, и было бы полезно знать, насколько сложным будет программное обеспечение еще на фазе проектирования.

Основываясь на своей предыдущей работе, которая демонстрировала, что повышение сложности программного обеспечения приводит к снижению его качества, они полагали, что если можно изменить сложность и управлять ею, можно улучшить программное обеспечение. Они были знакомы с работой Маккейба (McCabe)[6] по измерению цикломатической сложности и ее отношения к качеству программного обеспечения (опубликованной в 1976 году, на которую я ссылался ранее), а также с различными лексическими аналитическими методами. Они были убеждены, что могут получить измерение сложности кода, измеряя глубинную структуру кода вместо простого подсчета символов.

Так, используя теорию Маккейба и др., а также анализируя потоки между функциями и модулями, они предложили следующую формулу для измерения сложности:

$$(fan\_in * fan\_out)^2$$

Затем они вычислили это значение в рамках анализа кода операционной системы Unix и обнаружили 98%-ую корреляцию между их значением и "степенью изменения программного обеспечения" (т.е. исправления ошибок). Другими словами, они получили подтверж-

6 Рекомендуем ресурс http://www.mccabe.com/ для знакомства (прим. ред.).

дение, что чем сложнее функция или модуль, измеренный их формулой, тем больше вероятность ошибки в этой функции или модуле.

Дадим соответствующее формальное определения:

Разветвление на выходе процедуры A — это число локальных потоков из процедуры A плюс число структур данных, которые процедура A обновляет.

В этом определении поток A считается локальным:

1. Если A вызывает некий поток B.

2. Если некий поток B вызывает A и A возвращает значение в B, которое впоследствии используется B.

3. Если некий поток C вызывает два потока A и B, передавая результат (выходное значение) потока A в поток B.

Так, добавив все потоки для функции A, плюс число глобальных структур (внешних к A), которые A обновляет, мы получим разветвление на выходе для функции A. Разветвление на входе определяется по тому же принципу, в чем мы вскоре и убедимся (см. следующий раздел этой главы).

Согласно вышеприведенной формуле, для вычисления сложности кода вам нужно умножить значение разветвления на входе на разветвление на выходе, а затем возвести в квадрат полученное значение.

Используя эту меру, Генри и Кэфура отмечают три проблемы, свойственные очень сложному коду, которые данная мера позволяет выявить. Во-первых, высокое разветвление на входе и на выходе могут помочь найти функцию, которая пытается сделать слишком много. Эту функцию нужно разбить на несколько мелких. Во-вторых, высокие значения разветвлений на входе и выходе указывают на «точки стресса» в системе. Поддержка этих функций будет слишком сложной, поскольку они касаются многих других частей системы. И в-третьих, высокие значения этих параметров говорят о том,

что функция «недостаточно грамотно и чисто спроектирована», что означает, что она должна быть пересмотрена, подвержена рефакторингу, на фоне того, что она слишком большая и пытается сделать слишком много и у нее, скорее всего, есть отсутствующий уровень абстракции, что и является причиной высоких значений разветвления на входе и выходе.

Основываясь на измерениях сложности функций, Генри и Кэфура отмечают, что можно генерировать значения сложности модуля и оттуда можно измерить связь между самими модулями. Можно определить, какие модули имеют высокую сложность и должны быть пересмотрены, или же это свидетельствует о том, что необходим другой уровень абстракции. Внешним признаком слишком высокой сложности модуля может быть небольшое количество функций (например, три), но трактовка значения зависит от самого модуля. Когда модуль маленький и в нем 3 функции – это может быть хорошо, а когда модуль большой, а весь его код разбит всего на 3 функции – это плохо.

Генри и Кэфура исследовали длину функции и обнаружили, что всего у 28% функций, содержащих меньше 20 строк кода, были ошибки, тогда как в функциях, содержащих более 20 строк кода, были ошибки в 78% случаев. Остерегайтесь больших функций! Пусть код будет компактным!

Вернемся к определению разветвления на выходе и поговорим об его определяющих параметрах: данных и локальных потоков. Начнем с данных. В JavaScript достаточно просто объявить и использовать глобальные переменные, но стандарт JavaScript рекомендует нам не использовать глобальное пространство переменных и вместо этого использовать только локальные переменные. Это помогает уменьшать разветвление на выходе, поскольку мы уменьшаем число обновляемых структур данных.

Теперь о локальных потоках наших функций. Вы можете легко узнать число этих потоков, подсчитав количество инородных объектов, которые требует ваша функция. Следующий пример использует асинхронный механизм YUI, чтобы узнать зависимости кода для модуля `myModule`. Для интересующихся подробно загрузчик YUI описан по адресу http://yuilibrary.com/yui/docs/yui/loader.html. Здесь же нам нужно понимать только то, что мы просто гово-

рим фреймворку YUI о зависимости нашего кода, который он получит от нас. Затем он вызовет нашу `callback-функцию`, когда зависимость будет загружена:

```
YUI.use('myModule', function(Y) {
 var myModule = function() {
        this.a = new Y.A();
        this.b = new Y.B();
        this.c = new Y.C();
 };
 Y.MyModule = myModule;
}, { requires: [ 'a', 'b', 'c' ] });
```

Разветвление на выходе для конструктора `myModule` равно 3 (в нашем случае объекты даже не используются, они просто созданы и сохранены для будущих методов, но они все еще требуются, поэтому увеличивают число разветвления на выходе конструктора). Три неплохое число, но оно вырастет, когда внешний объект инстанцирует `myModule` и использует любое возвращаемое значение любого метода `myModule`.

Разветвление на выходе — мера, о которой вы должны помнить при редактировании метода. Это количество внешних методов и объектов, которыми этот метод манипулирует.

Независимо от локальных потоков закон Миллера утверждает, что задача запомнить и отследить больше чем 7 объектов – для человека является очень сложной. Причем более поздние исследования снизили эту планку до 4. А поскольку мы пишем код, который потом должен поддерживаться, читаться людьми, то поэтому мы должны учитывать эту человеческую особенность. И хотя формально нет какого-то определенного числа, которое является индикатором, предельным значением для разветвления на выходе, но когда значение разветвления на выходе больше 7 (или даже 4), пора задуматься от рефакторинге.

С технической точки зрения высокое разветвление на выходе проблематично по большому числу причин: код более сложен, его трудно понять и протестировать; нужно использовать mock-объекты и заглушки (stub) для каждого модуля, что делает тестирование сложным. Кроме того, высокое разветвление на выходе является показа-

телем высокой связанности кода, которая делает функции и модули чрезмерно хрупкими.

Стратегия разработки с учетом разветвления на выходе заключается в создании объекта/модуля, который инкапсулирует в себе некоторые разветвленные модули, выставляя наружу только одну функцию.

Рассмотрим следующий код, разветвление на выходе для которого как минимум 8:

```
YUI.use('myModule', function(Y) {
  var myModule = function() {
          this.a = new Y.A();
          this.b = new Y.B();
          this.c = new Y.C();
          this.d = new Y.D();
          this.e = new Y.E();
          this.f = new Y.F();
          this.g = new Y.G();
          this.h = new Y.H();
  };
  Y.MyModule = myModule;
}, { requires: [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ] });
```

Поддержка и тестирование этого кода будет проблематичным, он нуждается в рефакторинге. Идея заключается в вынесении связанных модулей в отдельный модуль:

```
YUI.use('mySubModule', function(Y) {
  var mySubModule = function() {
          this.a = new Y.A();
          this.b = new Y.B();
          this.c = new Y.C();
          this.d = new Y.D();
  };
  mySubModule.prototype.getA = function() { return this.a; };
  mySubModule.prototype.getB = function() { return this.b; };
  mySubModule.prototype.getC = function() { return this.c; };
  mySubModule.prototype.getD = function() { return this.d; };
  Y.MySubModule = mySubModule;
}, { requires: [ 'a', 'b', 'c', 'd'] });
```

```
YUI.use('myModule', function(Y) {
 var myModule = function() {
        var sub = new Y.MySubModule();
        this.a = sub.getA();
        this.b = sub.getB();
        this.c = sub.getC();
        this.d = sub.getD();
        this.e = new Y.E();
        this.f = new Y.F();
        this.g = new Y.G();
        this.h = new Y.H();
 };
 Y.MyModule = myModule;
}, { requires: [ 'mySubModule', 'e', 'f', 'g', 'h' ] });
```

Здесь мы создали уровень абстракции между `MyModule` и модулями `a`, `b`, `c`, и `d` и уменьшили на три разветвление на выходе, но помогло ли это? Помогло, хотя и незначительно. Но даже этот топорный рефакторинг сделал `MyModule` легче в поддержке. Общее количество тестов, однако, немного увеличилось, поскольку теперь нужно протестировать `MySubModule`, но наша цель — не уменьшить общее число тестов, а сделать проще тестирование каждого отдельного модуля или функции.

Наш рефакторинг `MyModule`, призванный уменьшить разветвление на выходе, не идеален. Модули и объекты, рефакторинг которых нужно произвести, должны быть в некотором роде связаны, и новый модуль должен обеспечить более интеллектуальный интерфейс для базовых объектов. То есть подмодули будут предоставлены новому модулю с более простым и объединенным интерфейсом. Снова общее число тестов увеличится, но каждую отдельную часть по отдельности будет проще тестировать и поддерживать. Вот небольшой пример:

```
function makeChickenDinner(ingredients) {
 var chicken = new ChickenBreast()
        , oven = new ConventionalOven()
        , mixer = new Mixer()
        , dish = mixer.mix(chicken, ingredients)
 return oven.bake(dish, new FDegrees(350), new Timer("50 минут"));
}
var dinner = makeChickenDinner(ingredients);
```

Эта функция разветвляется как сумасшедшая: создает пять внешних объектов и вызывает два метода двух разных объектов. Эта функция сильно связана с пятью объектами. Тестирование этой функции затруднено, поскольку вы нуждаетесь в заглушках во всех объектах и запросах. Имитация (mocking) метода `mix` и метода `bake` объекта `oven` будет очень сложной, поскольку оба возвращают значения. Давайте посмотрим, на что мог бы быть похож модульный тест для этой функции:

```
describe("test make dinner", function() {
 // Mocks
        Food = function(obj) {};
        Food.prototype.attr = {};
        MixedFood = function(args) {
                var obj = Object.create(Food.prototype);
                obj.attr.isMixed = true; return obj;
        };
        CookedFood = function(dish) {
                var obj = Object.create(Food.prototype);
                obj.attr.isCooked = true; return obj;
        };
 FDegrees = function(temp) { this.temp = temp };
 Meal = function(dish) { this.dish = dish };
 Timer = function(timeSpec) { this.timeSpec = timeSpec; };
 ChickenBreast = function() {
        var obj = Object.create(Food.prototype);
        obj.attr.isChicken = true; return obj;
 };
 ConventionalOven = function() {
  this.bake = function(dish, degrees, timer) {
     return new CookedFood(dish, degrees, timer);
   };
 };
 Mixer = function() {
  this.mix = function(chicken, ingredients) {
   return new MixedFood(chicken, ingredients);
  };
 };
 Ingredients = function(ings) { this.ings = ings; };
  // конец Mocks
  it("cooked dinner", function() {
```

```
    this.addMatchers({
      toBeYummy: function(expected) {
        return this.actual.attr.isCooked
        && this.actual.attr.isMixed;
      }
    });
    var ingredients = new Ingredients(петрушка', 'соль')
      , dinner = makeChickenDinner(ingredients)
    ;
    expect(dinner).toBeYummy();
    });
  });
```

Столь много всего! И что мы фактически здесь тестируем? Мы должны имитировать все ссылающиеся объекты и все объекты, на которые ссылаются ссылающиеся объекты, а также создаем новый объект `Ingredients` для выполнения метода `makeChickenDinner`, который тогда инстанцирует и использует наши сымитированные иерархии объектов. Имитация и замена всех объектов — сложная работа, особенно учитывая, что нам нужно протестировать всего один метод. Давайте осуществим рефакторинг и сделаем код хорошим: поддерживаемым и тестируемым.

Итак, исходная функция нуждается в пяти объектах — `ChickenBreast`, `ConventionalOven`, `Mixer`, `FDegrees` и `Timer`, число 5 для значения разветвления на выходе вполне нормально. Однако, связанность всех этих объектов очень высока. Связанность и разветвление на выходе обычно идет рука об руку, но не всегда. Мы можем уменьшить связанность с помощью инъекции или обработки событий, а обработку на выходе можем уменьшить, используя шаблон проектирования «Фасад» (facade)[7], которым мы охватим сразу несколько объектов (в дальнейшем, имея в виду конструкцию, созданную на основе шаблона «Фасад», мы будем просто говорить фасад). Подробнее обо всем этом мы поговорим в следующей главе, а сейчас вернемся к примеру.

Сперва мы можем создать *фасад*, представляющий "духовку" (oven). Создав "духовку", нам нужно установить ее "температуру"

7 Фасад – структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

(temperature) и "таймер" (timer). Затем мы выполняем инъекцию фасада для уменьшения связанности:

```
function Cooker(oven) {
        this.oven = oven;
}
Cooker.prototype.bake = function(dish, deg, timer) {
        return this.oven.bake(dish, deg, timer);
};
Cooker.prototype.degrees_f = function(deg) {
        return new FDegrees(deg);
};
Cooker.prototype.timer = function(time) {
        return new Timer(time);
};
function makeChickenDinner(ingredients, cooker) {
 var chicken = new ChickenBreast()
        , mixer = new Mixer()
        , dish = mixer.mix(chicken, ingredients)
 return cooker.bake(dish
        , cooker.degrees_f(350)
        , cooker.timer("50 minutes")
  );
}
var cooker = new Cooker(new ConventionalOven())
        , dinner = makeChickenDinner(ingredients, cooker);
```

У `makeChickenDinner` теперь есть две сильно связанных зависимости: `ChickenBreast` и `Mixer`. Фасад обрабатывает "всю работу по кухне". Причем он не использует весь API, предоставляемый "духовкой", "градусником" и "таймером", а использует только то, что необходимо для выполнения задания. Мы распределили зависимости более равномерно, что упрощает поддержку кода функции, ее тестирование и уменьшает число разветвления на выходе. Наш новый код (после рефакторинга) модульного теста будет таким:

```
describe("test make dinner refactored", function() {
 // Имитации
 Food = function() {};
 Food.prototype.attr = {};
 MixedFood = function(args) {
  var obj = Object.create(Food.prototype);
```

```
  obj.attr.isMixed = true;
  return obj;
};
CookedFood = function(dish) {
  var obj = Object.create(Food.prototype);
  obj.attr.isCooked = true;
  return obj;
};
ChickenBreast = function() {
  var obj = Object.create(Food.prototype);
  obj.attr.isChicken = true;
  return obj;
};
Meal = function(dish) { this.dish = dish };
Mixer = function() {
  this.mix = function(chicken, ingredients) {
    return new MixedFood(chicken, ingredients);
  };
};
Ingredients = function(ings) { this.ings = ings; };
// конец имитаций
it("cooked dinner", function() {
  this.addMatchers({
    toBeYummy: function(expected) {
      return this.actual.attr.isCooked
       && this.actual.attr.isMixed;
    }
  });
 var ingredients = new Ingredients('parsley', 'salt')
        , MockedCooker = function() {};
 // Локальный (для этого теста) имитированный объект Cooker,
 // который фактически может сделать тестирование
 MockedCooker.prototype = {
   bake: function(food, deg, timer) {
     expect(food.attr.isMixed).toBeTruthy();
     food.attr.isCooked = true;
     return food
 }
   , degrees_f: function(temp) { expect(temp).toEqual(350); }
   , timer: function(time) {
```

```
            expect(time).toEqual('50 minutes');
        }
      };
      var cooker = new MockedCooker()
            , dinner = makeChickenDinner(ingredients, cooker)
      ;
      expect(dinner).toBeYummy();
      });
    });
```

Этот новый тестовый код избавился от нескольких сымитированных объектов и заменил их локальным «поддельным» объектом, специфичным для этого теста, и сам может выполнить некоторое реальное тестирование, используя `expects` внутри себя.

Теперь намного проще тестировать объект `Cooker`, поскольку функциональность его совсем невелика. Мы можем пойти дальше и выполнить инъекцию `FDegrees` и `Таймер` вместо того, чтобы убрать тесную связность с ними. И, конечно же, мы также можем ввести или создать другой фасад для `Chicken` и, возможно, для `Mixer`. Вот окончательный код метода-инъекции:

```
function makeChickenDinner(ingredients, cooker, chicken, mixer) {
  var dish = mixer.mix(chicken, ingredients);
  return cooker.bake(dish
          , cooker.degrees_f(350)
          , cooker.timer('50 minutes')
  );
}
```

Здесь были выполнены инъекции зависимостей (о том, что такое зависимости, мы вскоре поговорим) и приводится соответствующий код для их тестирования:

```
describe("test make dinner injected", function() {
  it("cooked dinner", function() {
    this.addMatchers({
      toBeYummy: function(expected) {
        return this.actual.attr.isCooked
          && this.actual.attr.isMixed;
      }
    });
  var ingredients = ['parsley', 'salt']
```

```
  , chicken = {}
  , mixer = {
    mix: function(chick, ings) {
                    expect(ingredients).toBe(ings);
                    expect(chicken).toBe(chick);
                    return { attr: { isMixed: true } };
    }
  }
  , MockedCooker = function() {}
;
MockedCooker.prototype = {
    bake: function(food, deg, timer) {
                    expect(food.attr.isMixed).toBeTruthy();
                    food.attr.isCooked = true;
                    return food
    }
    , degrees_f: function(temp) { expect(temp).toEqual(350); }
    , timer: function(time) {
      expect(time).toEqual('50 minutes');
    }
  };
  var cooker = new MockedCooker()
          , dinner = makeChickenDinner(ingredients, cooker
          , chicken, mixer)
  ;
           expect(dinner).toBeYummy();
  });
});
```

Тестовый код получил полный контроль над имитациями, переданными в него, позволяя производить обширное и гибкое тестирование. Вау!

2.6. Разветвление на входе

Разветвление на входе — это мера числа модулей или объектов, от которых прямо или косвенно зависит ваша функция.

Большинство из того, что мы говорили о разветвлении на выходе, также применимо и к разветвлению на входе (fan-in), но не все.

Оказывается, что большое разветвление на входе может быть очень даже полезным. Подумайте об общих элементах приложения: журналировании, служебных подпрограммах, проверке аутентификации и авторизации и т.д. Эти функции должны вызываться всеми другими модулями приложения.

Вам не нужно несколько функций журналирования, вызываемых в разных частях кода, — везде должна использоваться одна и та же функция журналирования. Это называется повторным использованием кода, и это очень и очень хорошо.

Разветвление на входе – хороший показатель повторного использования общих функций в коде. По сути дела, если у общей функции очень низкое разветвление на входе, нужно убедиться, что нет дублирования кода где-то в другом месте программы, которое препятствует повторному использованию кода.

В некоторых случаях, тем не менее, высокое разветвление на входе все-таки является плохим показателем: у редко используемых и неслужебных функций должно быть низкое разветвление на входе. У высоких (высших) уровней абстракции кода также должно быть низкое разветвление на входе (идеально, разветвление на входе должно быть 0 или 1). Эти высокоуровневые части кода не предназначены для использования другими частями кода, обычно такие участки кода запускаются всего из одного места в коде и их разветвление равно 1.

Рассмотрим формальное определение разветвления на входе:

Разветвление на входе для процедуры А – это число локальных потоков, ведущих в процедуру А плюс число структур данных, от которых процедура А получает информацию.

В заключение разговора о разветвлениях на входе/на выходе отметим, что главное правило работы с разветвлением на входе/выходе сводится к следующему: не пропустить часть кода, у которого есть большое разветвление на входе и большое разветвление на выходе, поскольку это существенно усложняет код.

Как уже было сказано, разветвление на входе помогает вам обнаружить повторное использование кода. А мы знаем, что повторное использование кода — это хорошо, плохо — рассеянное журналирование и отладка функций через ваш код. Учитесь централизовать совместно используемые функции (и модули)!

2.7. Связанность

Если разветвление на выходе подсчитывает число модулей и объектов, зависимых от какого-либо модуля или функции (для которых и определяется значение разветвления на выходе), то связанность призвана показать то, как все эти модули используются вместе.

Подмодули могут уменьшить абсолютное число разветвления на выходе, но они не уменьшают сумму связей между исходным модулем (фрагментом кода) и необходимыми ему другими фрагментами кода (независимо от того, оформлен он в виде подмодуля или внешней библиотеки). Необходимость никуда не исчезла. Просто благодаря контролю разветвления на выходе можно локализовать взаимосвязи до определенных пределов.

Существуют подходы, метрики, которые пытаются выразить связанность в виде одного числа. Основываются эти подходы на шести уровнях, определенных Норманом Фэнтоном и Шари Лоуренс Пфлиджер в их работе «Software Metrics: A Rigorous & Practical Approach, 2nd Edition» (Course Technology), еще в 1996 г. При этом каждый уровень характеризуется своим числом — чем оно выше, тем сложнее связи. Далее мы рассмотрим эти шесть уровней (от самого сложного до самого простого) в последующих подразделах.

Связанность на уровне содержимого

Связанность на уровне содержимого – самая сложная форма связывания, подразумевающая вызовы методов или функций внешнего объекта и/или непосредственно изменяющая его путем редактирования свойств внешнего объекта. Пример:

```
Obj.property = 'бла';   // непосредственное изменение свойств объекта Obj
```

```
// Изменение методов Obj
Obj.method = function() { /* код */ };
Obj.prototype.method = function() { /* код*/ };
```

Все эти операторы иллюстрируют связанность на уровне содержимого нашего текущего объекта с объектом `Obj`. Этому виду связанности присвоено наивысшее число 5. То есть когда говорят, что связанность равна 5, то это значит, что там взаимное проникновение и связи на уровне содержимого.

Общая связанность

Немного ниже находится уровень общей связанности. **Ваш объект связан этим типом связывания с другим объектом, если оба объекта совместно используют глобальную переменную:**

```
var Global = 'global';
Function A() { Global = 'A'; };
Function B() { Global = 'B'; };
```

Здесь объекты `A` и `B` связаны общей связанностью. Данному типу связанности назначено число 4.

Связанность на уровне управления

Следующий уровень — связанность на уровне управления, немного более свободная форма связи, чем общая связанность. **Этот тип связи имеет место там, где внешний объект действует на основании флага или какого-либо параметра вашего объекта.** Например, создание одиночной абстрактной фабрики в начале вашего кода с передачей ей флага `env` — это и есть пример формы связанности на уровне управления:

```
var absFactory = new AbstractFactory({ env: 'TEST' });
```

Данному уровню назначено число 3.

Связанность по отпечатку в структуре данных (stamp coupling)

Связанность по отпечатку — это когда модули делят между собой составную структуру данных и каждый использует только её часть, по возможности даже не одну и ту же часть:

```
// Этот объект связан по отпечатку с объектом O
O.makeBread( { type: wheat, size: 99, name: 'foo' } );
// Где-то за пределами определения :
O.prototype.makeBread = function(args) {
        return new Bread(args.type, args.size);
}
```

Здесь мы передаем запись функции `makeBread`, но эта функция использует только два из трех свойств записи. Это и есть связанность по отпечатку. Данному уровню назначено число 2.

Связанность на уровне данных

Самая свободная связь их всех — связанность на уровне данных. **Этот тип связанности имеет место, когда объекты обмениваются сообщениями друг с другом без передачи управления.** Более подробно данный тип связанности мы изучим в главе 3, когда будем говорить о событийно-ориентированной архитектуре приложений. Данному уровню связанности назначено число 1.

Нет связанности

Последняя форма связанности (и лично моя самая любимая) с нулевой связью между двумя объектами, то есть когда вообще нет никакой связи между объектами. Этому уровню соответствует число 0.

Инстанцирование

Хотя формально инстанциирование[8] не относится к связыванию, действие инстанциирования глобального класса, не являющегося синглетоном (singleton)[9], является также очень сложной формой

8 **Инстанцирование (англ. instantiation)** — создание экземпляра класса. В отличие от слова «создание», применяется не к объекту, а к классу. То есть говорят: (в виртуальной среде) создать экземпляр класса или, другими словами, инстанцировать класс. Порождающие шаблоны используют полиморфное инстанцирование.

9 **Одиночка или синглетон (англ. Singleton)** — порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении у класса будет только один экземпляр класса, и предоставляющий к нему глобальную точку доступа. Существенно то, что можно пользоваться именно экземпляром класса, так как при этом во многих случаях становится доступной более широкая функциональность. Например, к описанным компонентам класса можно обращаться через интерфейс, если такая возможность поддерживается языком.

связывания, даже более сложной, чем общее связывание. Используя `new` или `Object.create`, вы можете создать одностороннее, сильно связанное отношение между объектами. То, что создатель делает с этим объектом, определяет, будет ли отношение двусторонним.

Инстанциирование объекта (создание экземпляра класса) делает ваш код ответственным за жизненный цикл другого объекта. В частности, раз объект был создан вашим кодом, то ваш код отвечает и за его уничтожение. Ведь даже если вы уже не используете этот объект, то он все равно может использовать память или даже выполняться. Таким образом, инстанциирование объекта навязывает создателю объекта определенные обязанности, о которых вы должны знать. Ну и конечно, чем меньше инстанциированных вашим модулем объектов, тем чище ваша совесть как разработчика. Уменьшение числа инстанциирований объектов минимизирует сложность кода, и к этому нужно стремиться. Если вы создаете слишком много объектов, пора остановиться и заново продумать вашу архитектуру.

Метрики связанности

Система именования и присвоения чисел разным типам/уровням связывания, рассмотренная нами ранее, лежит в основе создания совокупной базы, фрейма метрик связности в функциях, объектах и модулях по всему приложению. Одна метрика вычисляет связь между двумя модулями или объектами путем простого сложения числа соединений между модулями или объектами[10]. Другие метрики пытаются измерить связанность внутри одного модуля[11]. Также может быть создана матрица между всеми модулями приложения, чтобы просмотреть полную связь между каждым из них[12].

В результате формируется взгляд на систему с позиции связности – на основании чисел или наборов чисел, которые могут быть получены для определения того, как сильно или слабо связана система (элементы системы в рамках системы) или набор модулей. Этим взглядом мы должны осматривать свою систему, свой код каждый день.

10 Fenton, Norman, and Austin Melton. 1990. "Deriving Structurally Based Software Measures." Journal of Systems and Software12(3): pp. 177–187.

11 Dhama, Harpal. 1995. "Quantitative Models of Cohesion and Coupling in Software." Journal of Systems and Software29(1): pp. 65–74.

12 Alghamdi, Jarallah. 2007. "Measuring Software Coupling." Proceedings of the 6Th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, Corfu Island, Greece, February 16–19, 2007.

На основании этой информации мы можем при необходимости осуществить рефакторинг системы. В то же время необходимо понимать, что это за числа, не полагаться всецело на них (и на утилиты вычисления метрик связности) и параллельно с ними осуществлять самостоятельную инспекцию и ревью своего кода.

Связанность в реальном мире

Давайте рассмотрим несколько примеров связанности в JavaScript. Начнем с сильной связанности, так как это не только иллюстрирует саму сильную связанность как таковую, но и показывает слабую связанность, так как по опыту лучший способ понять, что такое слабая связанность, – это посмотреть на сильную связанность. Итак, смотрим пример:

```
function setTable() {
  var cloth = new TableCloth()
            , dishes = new Dishes();
  this.placeTableCloth(cloth);
  this.placeDishes(dishes);
}
```

Данный вспомогательный метод, вероятно, принадлежащий классу `Table`, пытается, так сказать, «аккуратно накрыть стол». Однако этот метод жестко связан с объектами `TableCloth` и `Dished`. Создание новых объектов внутри методов создает жесткую, сильную связь.

Этот метод стал не изолируемым из-за жесткой связи, в результате, когда нам нужно протестировать его или сделать еще что-то с ним, нам для этого обязательно понадобятся и объекты `TableCloth` и `Dishes`. В случае с тестированием модульные тесты будут стремиться протестировать метод `setTable` в изоляции от внешних зависимостей, но вышеприведенный код делает данную задачу очень сложной. И это несмотря на наличие в JavaScript возможностей выполнить динамическую инъекцию имитаций (mocks) и заглушек (stub), как это будет показано далее в этой книге. В данном примере имитировать объекты `TableCloth` и `Dished` будет мучительно непросто.

Мы можем позаимствовать некоторые идеи из какого-либо статически типизированного языка и использовать инъекцию (injection) для ослабления связи:

```
function setTable(cloth, dishes) {
        this.placeTableCloth(cloth);
        this.placeDishes(dishes);
}
```

Теперь наш метод стало намного проще изолировать и поэтому его намного проще протестировать: тестовый код может быть передан в имитациях и заглушках непосредственно в метод.

Однако в некоторых случаях подобный подход просто отодвинет проблему немного дальше, не избавившись от нее. Допустим, что у нас возникла необходимость инстанциировать некоторые объекты, в которых мы нуждаемся. Не будут ли эти методы сильно связаны?

Обычно мы хотим инстанциировать объекты как можно раньше и как можно выше в последовательности вызовов методов:

```
function dinnerParty(guests) {
  var table = new Table()
        , invitations = new Invitations()
        , food = new Ingredients()
        , chef = new Chef()
        , staff = new Staff()
        , cloth = new FancyTableClothWithFringes()
        , dishes = new ChinaWithBlueBorders()
        , dinner;
  invitations.invite(guests);
  table.setTable(cloth, dishes);
  dinner = chef.cook(ingredients);
  staff.serve(dinner);
}
```

А что, если теперь мы захотим сделать обед более неформальным и использовать бумажную скатерть и бумажные тарелки?

Как уже было сказано, идеально, чтобы все наши основные объекты были созданы в начале нашего приложения, но это не всегда возможно. Для таких ситуаций мы можем позаимствовать другой шаблон из статически типизированных языков: фабрики.

Фабрики позволяют нам инстанциировать объекты ниже по стеку вызовов, эти объекты получаются все еще связанными, но связь эта значительно слабее. Вместо явной зависимости от фактического объекта у нас теперь есть только зависимость от фабрики. У фабрики есть зависимости от объектов, которые она создает; однако для тестирования мы представляем фабрику, которая создает mock-объекты (объекты-имитаторы) или заглушки, а не реальные объекты. Таким образом, мы приходим к выводу, что нам нужны абстрактные фабрики.

Давайте начнем с обычной фабрики:

```
var TableClothFactory = {
  getTableCloth: function(color) {
    return Object.create(TableCloth,
          { color: { value: color }});
  }
};
```

Я параметризовал цвет скатерти в списке параметров нашей фабрики, чтобы получить новый экземпляр, который зеркально отражает список параметров конструктора. Использование этой фабрики довольно просто:

```
var tc = TableClothFactory.getTableCloth('purple');
```

Для тестирования нам не нужен реальный объект `TableCloth`. При желании протестировать наш код изолированно мы вместо реального объекта используем имитацию или заглушку:

```
var TableClothTestFactory = {
  getTableCloth: function(color) {
    return Y.Mock(); // или что-то еще
};
```

Здесь я использую фреймворк YUI, который будет подробно рассмотрен в главе 4. Код фабрики для получения сымитированной версии объекта `TableCloth` таков:

```
var tc = TableClothTestFactory.getTableCloth('purple');
```

Теперь, вот если бы *только* был *некоторый способ* согласовать вышеприведенные две фабрики так, чтобы наш код всегда правильно работал... Конечно, такой способ есть: все, что мы должны сделать,

так это создать фабрику, чтобы сгенерировать надлежащую фабрику `TableCloth`:

```
var AbstractTableClothFactory = {
  getFactory: function(kind) {
    if (kind !== 'TEST') {
      return TableClothFactory;
    } else {
      return TableClothTestFactory;
    }
  }
};
```

Все, что мы сделали, — это параметризовали фабрику так, чтобы она возвратила фактическую фабрику, которая возвращает требуемый вид объекта `TableCloth`. Этого монстра и называют абстрактной фабрикой. Вот как использовать ее в тесте:

```
var tcFactory = AbstractTableClothFactory.getFactory('TEST')
  , tc = tcFactory.getTableCloth('purple');
// мы получили mock-объект
```

Теперь у нас есть способ инстанциирования объектов без жесткой связи между ними. Мы отказались от сильных связей на уровне содержимого в пользу более слабой связи уровня управления. Теперь тестирование стало намного проще, поскольку мы можем создать версии фабрик, которые возвращают имитированные версии объектов вместо реальных, что позволяет нам протестировать код, не волнуясь обо всех его зависимостях. Код стал лучше.

Тестирование связанного кода

Понятно, что чем более сильно связан код, тем больше ресурсов нужно для его сопровождения и тестирования. Давайте пройдемся по уровням связывания и взглянем на них через призму тестирования.

Код, связанный на уровне содержимого, трудно протестировать, потому что модульное тестирование стремится тестировать код в изоляции. Но по определению код, связанный на уровне содержимого, сильно связан как минимум с одним другим внешним объектом. Вам потребуется полный диапазон

ухищрений модульного тестирования, чтобы попытаться протестировать такой код: скорее всего, придется масштабно использовать mock-объекты (объекты-имитации) и заглушки для репликации среды, в которой этот код может работать. Из-за жесткой сильной связи также необходимо интеграционное тестирование, чтобы убедиться, что все связанные объекты корректно работают вместе.

Код с общим связыванием проще тестировать, поскольку общие глобальные переменные можно легко имитировать или временно «заглушить» их с целью тестирования объекта.

Код со связыванием на уровне управления требует имитирования контролируемого внешнего объекта и проверки, правильно ли он управляется, что сделать достаточно просто, используя объект-имитацию (mock-объект).

При использовании связанности по отпечатку модуль легко протестировать путем имитации внешнего объекта и проверки, что переданный параметр был корректен.

Если используется связывание данных, то код можно очень легко протестировать посредством модульного тестирования. При полном же отсутствии связывания, у вас вообще идеальная картина для тестирования, так как код может отправляться на тест в том виде, в каком он есть: без всяких «костылей» типа объектов-имитаций, заглушек и т.п.

Внедрение зависимостей

Ранее мы уже вкратце говорили о внедрении (инъекции) зависимостей, теперь мы рассмотрим их поподробнее. Зависимости делают код сложным. Больше зависимостей – сложнее код, сложнее код – сложнее его понимать, сложнее его протестировать и сложнее отладить. Все, что вам хотелось бы упростить, зависимости усложняют.

Кроме того, по мере роста вашего приложения управление зависимостями кода занимает все больше и больше времени.

Между инъекцией и имитацией существует нестрогая взаимосвязь. Инъекция имеет дело с созданием и встраиванием объектов в код; имитация же представляет собой замещение объектов или вызовов методов заранее подготовленными заглушками (подставными версиями) с целью тестирования. Вы можете (и должны!) вставлять в код mock-объекты (имитирующие версии объектов) на этапе тестирования. Однако ни в коем случае не используйте фреймворк имитации в окончательной версии программного продукта!

Факторинг зависимостей или ручное введение их в конструкторы или вызовы методов помогает уменьшить сложность кода, но это также добавляет некоторые издержки: если была выполнена инъекция зависимостей объекта, другой объект(-ы) теперь ответственен за корректное построение этого объекта. Разве мы просто не передвигаем проблему на другой уровень? Да, так и есть!

С помощью внедрения зависимости мы можем внедрить полностью сформированные объекты в ваш код. При этом, мы должны указать инжектору, как фактически надо создавать эти объекты, и позже, когда вы захотите получить экземпляр такого объекта, инжектор возвратит вам один из них. Объекты не берутся из ниоткуда волшебным образом. Инжектор может создать только определенные вами объекты. При этом инжекторы предоставляют много разных способов описания того, как должны быть созданы объекты.

Вместе с объектом инжекторы обрабатывают контекст. Контекст сообщает инжектору, нужно ли создать новый экземпляр объекта или использовать существующий. Вы говорите инжектору, какой контекст будет у объекта, и впоследствии, когда ваш код запросит этот объект, инжектор выполнит правильное действие (или создаст новый экземпляр, или использует уже существующий).

В Java все это выполняется с помощью подклассов и интерфейсов, инжектор предоставляет конкретную инстанцию класса или подкласса. Эти типы могут быть проверены во время компиляции, и инжекторы могут принять меры против передачи null-объектов. В

JavaScript тоже есть оператор `instanceof`, но у него нет ни проверки типа, ни понятия реализации интерфейса. Так можем мы использовать внедрение зависимости в JavaScript или нет? Конечно, да!

Давайте вкратце рассмотрим `knit`, инжектор в стиле Google Guike для JavaScript. Рассмотрим следующий исходный код:

```
var SpaceShuttle = function() {
        this.mainEngine = new SpaceShuttleMainEngine();
        this.boosterEngine1 = new SpaceShuttleSolidRocketBooster();
        this.boosterEngine2 = new SpaceShuttleSolidRocketBooster();
        this.arm = new ShuttleRemoteManipulatorSystem();
};
```

Со всеми объектами, инстанциируемыми в конструкторе, как мы можем протестировать `SpaceShuttle` без его «двигателей» и «пушки»? Инстанциирование зависимых объектов в конструкторах (или в другом месте в объекте) сильно связывает зависимость, которая делает тестирование более трудным, а значит нужно поручить это инжектору.

Первым делом нужно сделать конструктор инъекцируемым:

```
var SpaceShuttle = function(mainEngine, b1, b2, arm) {
        this.mainEngine = mainEngine;
        this.boosterEngine1 = b1;
        this.boosterEngine2 = b2;
        this.arm = arm;
};
```

Теперь мы можем использовать `knit` для определения того, как мы хотим, чтобы создавались наши объекты:

```
knit = require('knit');
knit.config(function (bind) {
        bind('MainEngine').to(SpaceShuttleMainEngine).is("construtor");
        bind('BoosterEngine1').to(SpaceShuttleSolidRocketBooster)
                .is("constructor");
        bind('BoosterEngine2').to(SpaceShuttleSolidRocketBooster)
                .is("constructor");
        bind('Arm').to(ShuttleRemoteManipulatorSystem).is("constructor");
        bind('ShuttleDiscovery').to(SpaceShuttle).is("constructor");
        bind('ShuttleEndeavor').to(SpaceShuttle).is("constructor");
```

```
        bind('Pad').to(new LaunchPad()).is("singleton");
});
```

Здесь объекты `SpaceShuttleMainEngine`, `SpaceShuttleSolidRocketBooster` и `ShuttleRemoteManipulatorSystem` определены в другом месте:

```
var SpaceShuttleMainEngine = function() {
  ...
};
```

Теперь каждый раз, когда потребуется `MainEngine`, инжектор `knit` заполнит его:

```
var SpaceShuttle = function(MainEngine
        , BoosterEngine1
        , BoosterEngine2
        , Arm) {
        this.mainEngine = MainEngine;

}
```

Таким образом, весь объект `SpaceShuttle` со всеми его зависимостями доступен в методе `knit.inject`:

```
knit.inject(function(ShuttleDiscovery, ShuttleEndeavor, Pad) {
        ShuttleDiscovery.blastOff(Pad);
        ShuttleEndeavor.blastOff(Pad);
});
```

Инжектор `knit` рекурсивно выясняет все зависимости `SpaceShuttle` и создает объекты `SpaceShuttle` для нас. Определение `Pad` как синглетона (одиночного элемента, singleton) гарантирует, что любой запрос объекта `Pad` будет всегда возвращать одно инстанциирование.

Предположим, со временем «Мексика» создает еще более совершенный объект `ShuttleRemoteManipulatorSystem`, чем «Канада». Переключение на этот тип «оружия» происходит очень просто:

```
bind('Arm').to(MexicanShuttleRemoteManipulatorSystem).is("constructor");
```

Теперь все объекты, требующие `Arm`, получат мексиканскую версию вместо канадской без каких-либо других изменений в коде.

Помимо выгрузки объектов фреймворк инжектора позволяет также вставить имитированные или тестовые объекты в вашем приложении, просто изменив привязку. В нашем примере это может быть полезно для тестирования `SpaceShuttle`.

В заключение данного раздела отмечу существование фреймворка **AngularJS** (http://angularjs.org/), который позволяет внедрять зависимости через регулярные выражения. Как следствие, помимо упрощения тестирования, контроллеры и другие части функционала могут определять, какие объекты необходимы, чтобы выполнить их работу (по списку функций), а затем внедрять надлежащий объект.

2.8. Комментарии

В идеальном JavaScript-коде обязательно присутствует комментарий перед каждой функцией и методом. Ведь при поддержке кода именно там первым делом читают блоки комментариев (особенно перед публичными функциями) — они сообщают программисту всю необходимую информацию. Написание эффективных комментариев и их совершенствование является неотъемлемой частью работы разработчика.

Часто комментариями пренебрегают на фоне написания тестов. Однако чтение кода — самый прямой способ понять его и протестировать, чего не скажешь о тестовых скриптах. Комментарии очень важны, особенно комментарии кода, сложного для восприятия. Важно, чтобы ваши комментарии объясняли, что делает метод и как он это делает.

Через шесть месяцев, вероятно, вы уже не будете смотреть на тот или иной фрагмент кода так, как вы смотрите на него сейчас, во время его создания. Скорее всего, вы уже даже и не вспомните, что и почему именно так вы сделали. Здесь на помощь придут хорошие комментарии. До сих пор мы говорили только о том, что делает код сложным и как уменьшить эту сложность. Но нет такого понятия, как нулевая сложность. Разработка программного обеспечения вообще сложна, по определению. А разработка хорошего программного обеспечения — еще сложнее. Однако вы можете упростить свой

код, точнее упростить его понимание с помощью написания комментариев.

Исследование Enerjy (http://www.enerjy.com/techpaper-2008-01-02.pdf) установило, что две из трех эмпирических метрик для определения «ненадежности» кода основываются на комментариях, а именно блоках комментария в начале функций. Отсутствующие или некорректные комментарии перед функциями являются индикаторами ненадежности кода. Казалось бы, причем тут комментарии и надежность? Однако статистика установила корреляцию между этими показателями. А логика проста: комментарии (корректные комментарии!) требуют полного понимания кода. Больше комментариев — больше понимания. Больше понимания – лучше качество и надежность кода. Вот и получается, что комментарии становятся индикатором того, что разработчик кода понимал, что делал, когда писал код программы.

Когда лучше писать комментарии? А когда вы только приступаете к написанию функции и у вас имеется полное понимание того, что делает или что должна делать функция. Это самое лучшее время, чтобы прокомментировать код. Постарайтесь даже написать комментарий раньше, чем вы начнете писать функцию. Вы также можете написать тест до написания самой функции, используя разработку через тестирование (TDD). При обслуживании функции вы также должны прокомментировать любые внесенные изменения. Комментарии вынуждают вас понять свой код. Они действуют как мини-самоанализ вашего кода. А надеюсь, не надо объяснять, насколько важно, чтобы вы сами понимали написанный вами код.

Что писать в комментариях? Блоки комментариев перед определением функции должны ясно дать понять, что будет передано функции в качестве списка параметров и что будет возвращено, а также что означают возвращаемые значения. Идеально, когда у функции нет побочных эффектов, но если они есть, они должны быть явно указаны в блоке комментария. Внутри методов вы должны использовать комментарии только для объяснения сложного кода.

Комментарии также делают код более тестируемым. В них, по сути, информация о том, что и как тестировать.

Далее, используя в своих интересах структурированные комментарии, вы можете легко преобразовать всю вашу работу в читаемый HTML-код. И сейчас мы рассмотрим несколько инструментов, которые позволяют генерировать документацию для кода на основе имеющихся комментариев.

Использование инструмента YUIDoc для генерации документации

YUIDoc (**http://yui.github.io/yuidoc/**) — это пакет Node.js, доступный через npm. Инсталляция предельно проста:

```
% npm -g install yuidocjs
```

После этого вам становится доступен исполнимый файл `yuidocjs`, предназначенный для преобразования всех ваших комментариев в симпатично оформленный HTML. В этом разделе мы обсудим последнюю на момент написания этих строк его версию — 0.6.0.

YUIDoc использует формат комментариев Java, где комментарии начинаются с /* и заканчиваются */. Но на практике вы можете запускать YUIDoc для любого языка программирования, где блоки комментариев начинаются с /* и заканчиваются */.

YUIDoc предоставляет семь основных тегов. Каждый блок комментариев должен содержать *один и только один из этих основных тегов*. Исчерпывающий список тегов доступен по адресу: **http://yui.github.io/yuidoc/syntax/index.html#primary-tags**. В то же время комментарий может содержать несколько вторичных тегов (см. далее).

Создание документации утилитой командной строки `yuidocjs` предельно просто. Однако нужно иметь в виду, что YUIDoc работает только с каталогами, поэтому, в отличие от JSDoc (мы его обсудим позже), вы не можете сгенерировать документацию для одного JavaScript-файла.

Типичная команда командной строки для создания документации выглядит примерно так:

```
% yuidoc -o yuidoc -c src/yuidoc.json src
```

Здесь `yuidoc` — имя выходного каталога, в который будут помещены все сгенерированные HTML-файлы, а `src` — корневой каталог ваших JavaScript-файлов, по которому рекурсивно пройдется **yuidoc** для создания документации. Параметр **-c** задает файл конфигурации `yuidoc.doc` в формате JSON. В нашем случае этот файл может быть пустым JSON-объектом:

```
% cat src/yuidoc.json
{}
```

Рассмотрим JavaScript-файл с комментариями, оформленными в соответствии с правилами (нотацией) YUIDoc:

```
/**
* Предоставляет некоторые математические функции
*
* @class Math
**/
/**
* Эта функция принимает два операнда, 'a' и 'b', и возвращает
* их сумму (или конкатенацию, если они являются строками)
*
* @method sum
* @param {Number or String} a первый операнд
* @param {Number or String} b второй операнд
* @return {Number or String} Сумма
*/
exports.sum = function(a, b) { return a + b };
/**
* Эта функция принимает два операнда, 'a' и 'b', и возвращает
* их произведение
*
* @method product
* @param {Number} a первый операнд
* @param {Number} b второй операнд
* @return {Number} Произведение
*/
exports.mult = function(a, b) { return a * b };
```

На основании данного JavaScript-кода YUIDoc сгенерирует страницу, подобную изображенной на рис. 2.2.

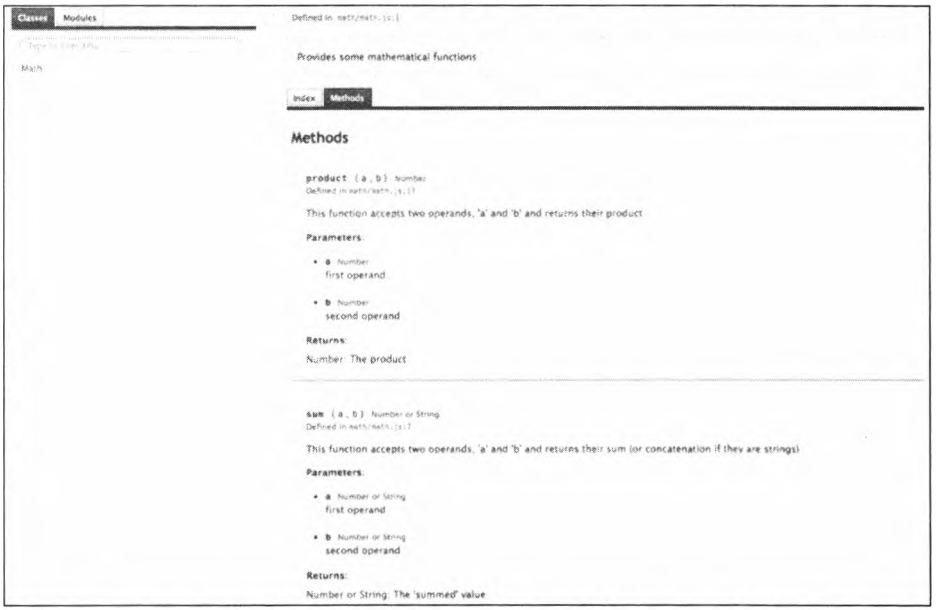


Рис. 2.2. Сгенерированный YUIDoc HTML-код

На мой взгляд, YUIDoc генерирует гораздо более привлекательные шаблоны (с возможностью поиска!), а его библиотека тегов более дружественна, чем в случае с JSDoc, с которым мы познакомимся в следующем разделе. Но выбор за вами. Наша задача состоит в том, чтобы дать обзор возможных решений.

Расширенные возможности генерации документации с помощью инструмента JSDoc

JSDoc (https://github.com/jsdoc/jsdoc) подобен YUIDoc, но обладает более обширным списком тегов (https://github.com/jsdoc/jsdoc).

Примечание.

Есть такой инструмент, как Google Closure Compiler (https://github.com/google/closure-compiler/wiki/Annotating-JavaScript-for-the-Closure-Compiler), который предназначен для минимизации, оптимизации и компиляции JavaScript-кода. Так вот он очень активно использует теги JSDoc, так что, изначально ориентируясь на оптимизацию с GCC, вы, используя теги JSDoc для комментирования, в итоге можете убить двух зайцев одним выстрелом.

JSDoc – это по сути Java-программа с немного аляповатой настройкой. Сама программа находится в JAR-файле + к нему ряд

дополнительных файлов шаблонов. После загрузки и распаковки пакета нужно установить две переменные окружения — `JSDOCDIR` и `JSDOCTEMPLATEDIR`. Переменная `JSDOCDIR` задает каталог, в котором находится файл `jsrun.jar`. `JSDOCTEMPLATEDIR` содержит имя каталога шаблонов и всегда должен указывать на каталог `$JSDOCDIR/templates/jsdoc`, если вы, конечно, не собираетесь создавать собственные шаблоны для JSDoc.

Как только вы установили эти две переменных окружения, вы можете использовать сценарий оболочки `jsrun.sh` (находится в каталоге `$JSOCDIR`) так:

```
% /bin/sh $JSDOCDIR/jsrun.sh  -d=<output dir> <JavaScript file>
```

Эта команда создаст набор HTML-файлов и поместит их в результирующий каталог `<output dir>`. Теперь рассмотрим JavaScript-файл с комментариями, подготовленными в стиле JSDoc (обратите внимание на разницу между JSDoc-тегами и YUIDoc-тегами):

```
/**
* Эта функция принимает два операнда, 'a' и 'b', и возвращает
* их сумму (или конкатенацию, если они являются строками)
*
* @name sum
* @function
* @param {Number or String} a первый операнд
* @param {Number or String} b второй операнд
* @returns {Number or String} Сумма
*/
exports.sum = function(a, b) { return a + b };
/**
* Эта функция принимает два операнда, 'a' и 'b', и возвращает
* их произведение
*
* @name product
* @function
* @param {Number} a первый операнд
* @param {Number} b второй операнд
* @returns {Number} Произведение
*/
exports.mult = function(a, b) { return a * b };
```

Затем запустите JSDOC следующим образом:

```
/bin/sh $JSDOCDIR/jsrun.sh -d=jsdoc mySum.js
```

Этот код создаст несколько HTML-файлов в каталоге `jsdoc`. Самый интересный HTML-файл выглядит подобно изображенному на рис. 2.3.

Вывод очень симпатичный, но не до такой степени, как в случае с YUIDoc.

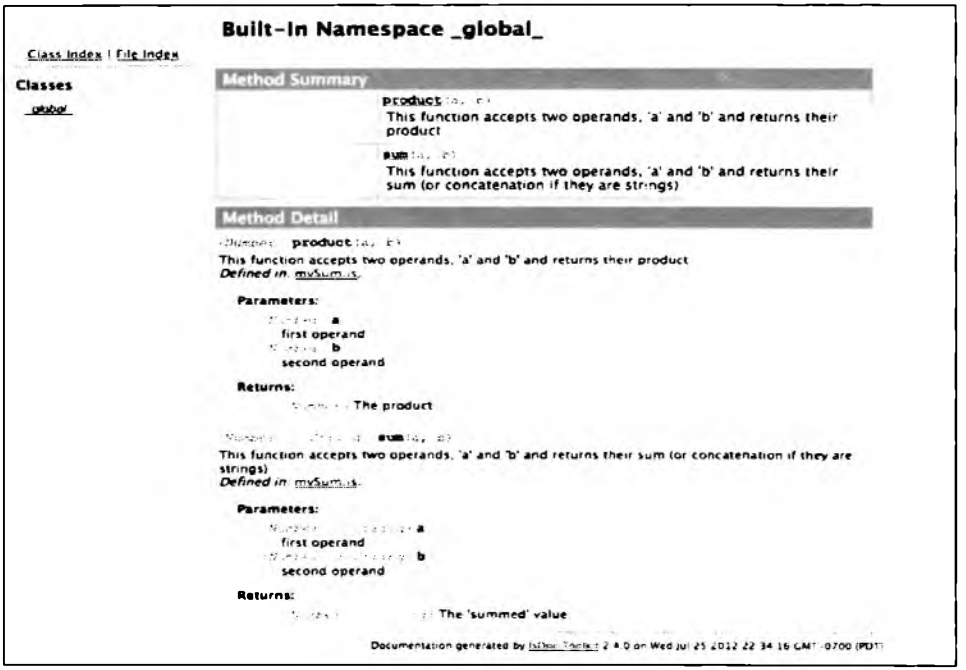


Рис. 2.3. HTML, сгенерированный директивами jsdoc

Генераторы документации Docco/Rocco

Как сказано на официальном сайте компании-производителя: "Docco — быстрый и "грамотный" генератор документации вашего кода".

Docco (http://jashkenas.github.io/docco/) — родитель целого семейства программ, которые делают одно и то же: извлекают комментарии, размеченные в стиле Markdown[13], из исходного кода и преоб-

13 Markdown (Маркдаун) – облегченный язык разметки.

разуют их в HTML. Лучше всего понять, как работает Docco, можно, посетив его сайт: веб-сайт Docco сгенерирован самим же Docco.

Rocco — это клон Docco, написанный на Ruby. Отличается от своего прародителя немного более интеллектуальным форматированием JavaScript-кода, чем Docco (который написан на CoffeeScript). Сайт Rocco (http://rtomayko.github.io/rocco/) тоже написан на Rocco.

Самое большое препятствие к использованию Rocco — необходимость установки Ruby для его выполнения. Следующее ограничение – **Pygments** – утилита подсветки синтаксиса, написанная на Python. Если эта утилита недоступна локально, Rocco попытается использовать веб-сервис Pygments, но я сомневаюсь, что вы захотите, чтобы написанный вами код гулял туда-сюда по сети для подсветки синтаксиса.

Pygments зависит от Python, поэтому, как только Python установлен, Pygments может быть установлен так:

```
% sudo easy_install Pygments
```

Rocco также нуждается в синтаксическом разборе Маркдауна (парсере Markdown), для чего вам нужно установить `rdiscount`:

```
% sudo gem install rdiscount
```

Наконец, можно установить и Rocco:

```
% sudo gem install rocco
```

Теперь просто запустим Rocco, «скормив» ему ваш код:

```
% rocco myJavascript.js
```

По умолчанию у результирующего файла будет такое же имя, что и у джаваскриптовского файла, но с расширением `.html`. Файл будет создан в текущем каталоге, но с помощью опции **-o** вы можете указать другой каталог, в котором должен будет сохранен результирующий HTML-код.

В результате использования Rocco все комментарии будут извлечены из исходного кода и помещены слева на результирующей вебстранице, сам комментируемый код будет помещен справа от комментариев. Рассмотрим следующий JavaScript-код:

```
/**
* Эта функция принимает два операнда, 'a' и 'b', и возвращает
* их сумму (или конкатенацию, если они являются строками)
*/
exports.sum = function(a, b) { return a + b };
/**
* Эта функция принимает два операнда, 'a' и 'b', и возвращает
* их произведение
*/
exports.mult = function(a, b) { return a * b };
```

Результат будет представлен на рис. 2.4.

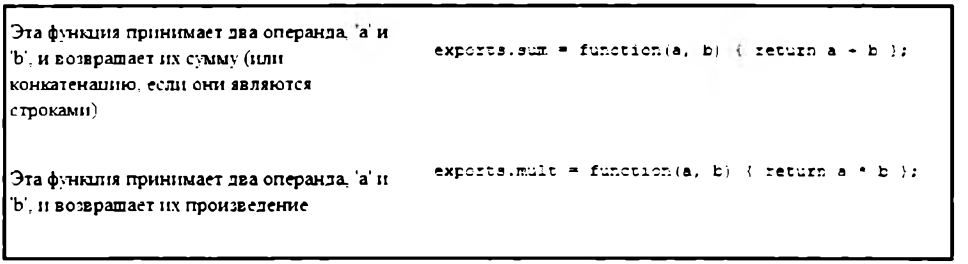


Рис. 2.4. HTML-код, сгенерированный Rocco

Довольно приятно выглядит! Таким образом, Rocco и компания предоставляют немного иной взгляд на ваш код и методику его документирования.

2.9. Тестирование человеком: человеческий тест, размер кода и инспекция кода по Фагану

Ну и, наконец, один из самых лучших инструментов для определения сложности кода – это ваш коллега. Просто покажите ему (или ей) ваш код и посмотрите на реакцию. Смог ли он понять код и то, как он работает? Двадцать минут — это верхняя граница концентрации обычного взрослого человека. Если вашему коллеге требуется больше 20 минут, чтобы понять суть вашего кода, — не все приложе-

ние, а именно метод или объект, абстрагированный от всего вашего приложения, — что-то в вашем коде неправильно.

Это мы рассмотрели, так сказать, неформальный процесс инспекции вашего кода каким-то внешним лицом, однако существуют и формализованные методики инспекции/контроля качества кода с привлечением других людей в качестве инспекторов. Наиболее развитой и известной методикой такого рода является так называемая «Инспекция программного кода по Фагану» (http://www.mfagan.com/pdfs/aisi1986.pdf). Формальный процесс, оформленный в виде методики и изученный Фаганом, обнаруживает от 60 до 90% всех дефектов программы. Это очень высокий процент. Процесс контроля Фагана довольно сложен и здесь мы его не будем рассматривать детально, но стоит отметить, что никакой другой метод даже близко не обеспечивает такой процент нахождения ошибок. Когда вы решите действительно серьезно относиться к нахождению ошибок, обратитесь к методике «Инспекции программного кода Фагана».

В то же время неформальные просмотры кода также дают превосходные результаты и являются хорошими инструментами для измерения сложности кода и его пригодности для обслуживания и тестируемости.

И напоследок еще несколько замечаний по существу:

Интересно, что операторы if без фигурных скобок — также по статистике являются показателем предрасположенности к ошибкам, поэтому не забывайте всегда использовать фигурные скобки!

Второе замечание касается размера кода. Размер кода работает против вас и со временем он только растет. При этом необходимо понимать, что уменьшение размера кода и максимизация его доступности не означают ограничение объема самого кода в вашем приложении. Это означает, что нужно ограничить объем кода в каждой функции, модуле и файле. У вас может быть много функций, модулей и файлов, но это нормально. Пока их отдельно взятые размеры разумны и они понятны, сложность кода будет минимальной.

Резюме

Сложность кода может проявляться в разных формах и в разных размерах. Большая часть сложности может быть измерена. Статический анализ кода поможет эффективно очистить ваш код, сделать его тестируемым и совершенным. Используя хорошо понятные шаблоны и опираясь на «чистый» стандарт JavaScript, вы сделаете свой код удобным в сопровождении.

Огромное влияние на сложность кода оказывают имеющиеся в коде зависимости. Они должны быть грамотно построены, по возможности разрушены, но в любом случае поняты и обслужены. Инстанциирование зависимостей — опасное дело. Если процесс может быть разгружен в инжектор, то это очень хорошо. Инстанциирование зависимых объектов образует сильные зависимости, что существенно усложняет тестирование, поэтому и необходима разгрузка процесса в инжектор — пусть инжектор сделает самое трудное за вас.

Написание хорошей документации в комментариях крайне важно для упрощения кода, по крайней мере, на уровне его восприятия и понимания происходящего. К тому же вы можете автоматически сгенерировать документацию по комментариям в коде, используя YUIDoc, JSDoc и Rocco, что, вообще говоря, существенно упростит создание документации с каждой новой сборкой кода. При всем при этом необходимо иметь в виду, что написание комментариев — это только половина работы. Еще более важно обновлять созданные комментарии по ходу развития и изменения проекта.

Глава 3.

Событийно-ориентированная архитектура

Использование «фабрик» различных абстракций, «фасадов» и других шаблонов проектирования – это не единственный способ разрушения зависимостей и достижения изолированности кода. Более JavaScript-ориентированное решение состоит в использовании событий. Функциональная природа JavaScript делает его идеальным языком для событийно-ориентированного программирования. Об этом мы и поговорим в данной главе.

3.1. Преимущества событийно-ориентированного программирования

Все приложения так или иначе используют систему передачи сообщений. Код приложения обменивается сообщениями с какими-то объектами (то есть приложение отправляет сообщение объекту и, возможно, ждет ответ на это сообщение), следовательно, у кода есть ссылка на эти объекты и говорят, что код жестко связан с данными объектами. Эти объекты могут быть либо глобальными, переданными в код, либо введенными в виде параметра функции, либо они создаются локально. В предыдущей главе мы говорили о локальном инстанциировании объектов и фабриках. Фабрики позволяют преобразовать сильную связанность в слабую: при использовании фабрик код зависит не от объектов, а от фабрики.

С локальным инстанциированием мы разобрались, но мы все еще имеем дело или с глобальными переменными или с внедренными зависимостями. Глобальные зависимости очень опасны. Они могут повлиять на любую часть системы, мы можем их случайно изменить, и в результате возникнут ошибки, найти которые будет очень

сложно. К тому же возможны потери данных, если, например, у нас появится локальная переменная с таким же именем, что существенно усложняет отладку кода.

В JavaScript объявление и использование глобальных переменных чрезвычайно просто, и окружение обычно уже предоставляет несколько глобальных переменных (например, объект `window` находится в глобальной области видимости). Также уже изначально доступно много глобальных функций и объектов (например, объект YUI или $ из jQuery). Это означает, что мы должны сделать все возможное, чтобы не ввести свои переменные в глобальное пространство имен, поскольку там уже и так много чего есть.

Если мы не хотим ни создавать объект локально, ни помещать его в глобальное пространство имен, нам остается использовать инъекцию. Инъекция — не панацея, однако теперь у нас должны быть функции установки для обработки инъекции и, что более важно, теперь мы должны иметь дело с фабриками или фреймворками внедрения зависимостей. В любом случае это шаблонный код, который не является специфичным для нашего приложения.

Проблема с зависимостями возникает, поскольку мы должны взаимодействовать с другим кодом, который может быть внутренним или внешним для нашего приложения. Возможно, нам нужно будет передать параметры этому коду. Возможно, нам нужно получить от него какой-то результат. Выполнение этого кода может занять много времени. Мы можем ждать завершения этого кода, прежде чем продолжить выполнение нашего приложения, или же мы можем продолжить выполнение, пока код будет выполняться в отдельном потоке.

Все это достигается с использованием прямых ссылок на другие объекты. Используя эти ссылки, мы вызываем методы или устанавливаем свойства объекта, а затем получаем результаты. Но событийно-ориентированное программирование предоставляет альтернативный способ передачи сообщения объектам. По своей природе использование событий не очень отличается от вызова метода через фреймворк обработки событий. У вас все еще должна быть локальная ссылка на объект, чтобы создать для него событие или же, наоборот, прослушать события, генерируемые объектом.

Примечательно, что у JavaScript нет формальной поддержки событий и обратных вызовов, исторически язык лишь предоставляет функции как объекты первого класса[1]. Это позволило JavaScript «столкнуть в кювет» интерфейсно-ориентированную модель событий Java, в которой все должно быть объектами, что приводит к довольно неуклюжему синтаксису. Модель событий, обеспечиваемая DOM в браузерах, добавила поддержку событий в JavaScript вплоть до используемого в Node.js механизма асинхронного программирования с широким использованием обратных вызовов.

Событийно-ориентированное программирование сводится к двум понятиям: вызову и возврату. Оно преобразует вызов в параметризованное событие, а возврат — в параметризованный обратный вызов (callback). Магия происходит, когда есть требования локальной ссылки сделать эти вызовы и возвраты абстрагированными, позволяя нам взаимодействовать с другим кодом без необходимости иметь локальную ссылку на него.

3.2. Концентратор событий

Что такое концентратор событий

Идея обработки событий очень проста.

Концентратор событий содержит информацию о методах, выступающих в роли обработчиков определенных событий. Концентратор событий — единственное централизованное место, в которое можно отправить запросы события и откуда нужно ждать ответа. Методы могут как обрабатывать, так и генерировать события. Методы получают асинхронные ответы через обратные вызовы. Классам или методам вашего приложения по большому счету нужна всего лишь одна ссылка — на концентратор событий. Все коммуникации обрабатываются концентратором. Код, выполняющийся в браузере или запущенный на сервере, имеет равный доступ к концентратору.

1 Объект называют «объектом первого класса», если он: может быть сохранен в переменной или структурах данных; может быть передан в функцию как аргумент; может быть возвращен из функции как результат; может быть создан во время выполнения программы; внутренне самоидентифицируем (независим от именования). Термин «объект» используется здесь в общем смысле и не ограничивается объектами языка. программирования. Так, значения простейших типов данных, например integer и float, во многих языках являются «объектами первого класса».

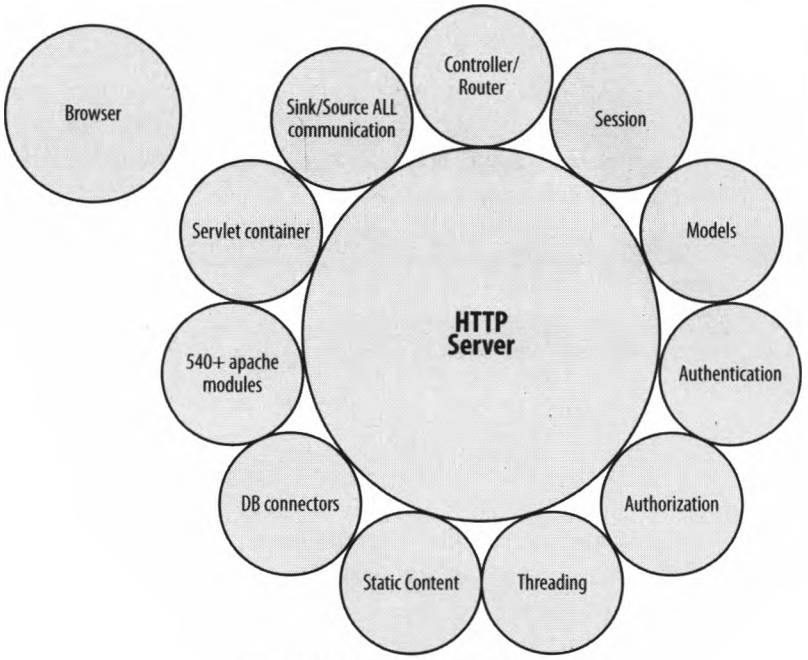


Рис. 3.1. Типичное веб-приложение

Типичное веб-приложение сильно связано с веб-сервером, что показано на рис. 3.1.

Как видно из рис. 3.1, браузеры и клиенты находятся в стороне, они соединяются с HTTP-сервером, который загружен различными сервисами и модулями. Все дополнительные сервисы сильно связаны с самим HTTP-сервером и предоставляют определенные услуги. Вся передача между клиентами и модулями происходит через HTTP-сервер, что не очень хорошо, поскольку по своей природе HTTP-серверы идеально подходят для обслуживания статического контента (изначально они для этого и были разработаны). Событийно-ориентированное программирование заменяет HTTP-сервер на концентратор событий.

Чтобы присоединиться к системе, все, что мы должны сделать, — просто добавить ссылку на концентратор, что и показано в следующем коде:

```
eventHub.fire(
        'LOG'
        , {
```

```
            severity: 'DEBUG'
            , message: "Я что-то делаю"
        }
);
```

Тем времени, где-то на сервере, возможно, есть следующий код, занимающийся обработкой события LOG:

```
// «Прослушиваем» событие LOG, как только оно возникнет,
// будет вызвана функция logIt
eventHub.listen('LOG', logIt);
// Функция-обработчик события
function logIt(what) {
        // делаем что-то интересное
        console.log(what.severity + ': ' + what.message);
}
```

Концентратор событий соединяет эти два несвязанных фрагмента кода. Сообщения передаются между клиентами концентратора события независимо от их расположения. Рис. 3.2 иллюстрирует отношения между концентратором и всеми его подключенными моду-

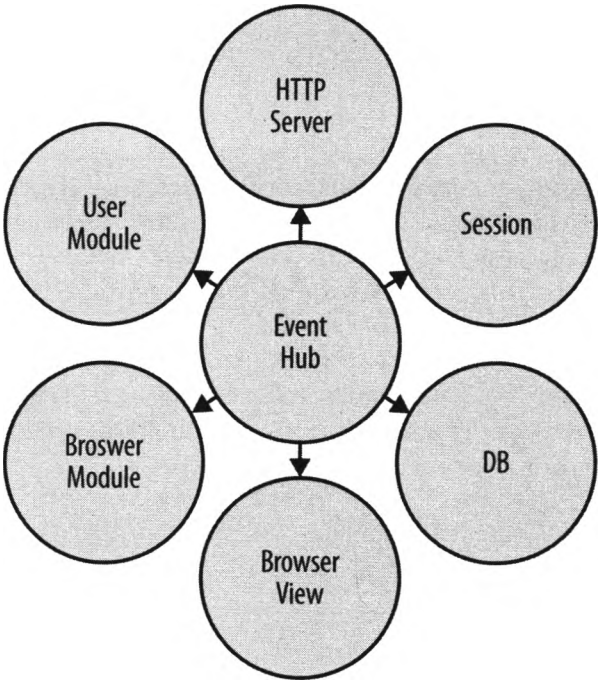


Рис. 3.2. Все клиенты (браузеры и клиенты стороны сервера) подключены к одному и тому же концентратору событий

лями. Обратите внимание на то, что HTTP-сервер теперь не "центр вселенной", а понижен до уровня модуля, предоставляющего какой-то сервис, подобно остальным подключенным модулям.

Но что же, теперь у нас появилась общая связанность? Как было сказано в главе 2, общая связанность – это один из самых сложных типов связанности, когда объекты разделяют общую глобальную переменную. Но у нас здесь все по-другому.

Независимо от того, является ли объект `EventHub` глобальной переменной или передан в каждый объект в качестве параметра, это не общая связанность. Да, все объекты используют общий объект `EventHub`, но они не изменяют его и не делятся информацией через него, изменяя его состояние. Это важное и интересное отличие `EventHub` от других видов глобальных объектов. Несмотря на то, что все коммуникации между объектами осуществляются через объект `EventHub`, в нем нет никакого общего ресурса.

Использование концентратора событий

Разберемся, что мы имеем. У нас есть фрагменты кода (модули), соединенные с концентратором, мы слушаем события и реагируем на них. Вместо вызова методов (что требует локального инстанцирования объекта, связанности, разветвления на выходе и т.д.) мы просто порождаем событие и, возможно, ожидаем ответа на него (если мы в этом нуждаемся).

Концентратор может работать везде, где вам захочется, – понятно, что на стороне сервера. Клиенты могут соединяться с ним, порождать события и подписываться на другие события.

Обычно клиенты стороны браузера порождают события, а клиенты стороны сервера прослушивают их, но, конечно же, любой клиент стороны сервера может породить событие и прослушать события других клиентов.

Эта архитектура использует сильные функциональные качества JavaScript и позволяет поддерживать код приложения в виде небольших фрагментов с минимальными зависимостями. А это очень хорошо! Использование событий вместо вызовов методов значительно упрощает тестирование и поддержку кода, поощряя разра-

ботчиков писать небольшие блоки кода с минимальными зависимостями. Вся функциональность, выходящая за пределы «компетенции» какого-либо модуля, предоставляется ему в виде сервиса вместо того, чтобы использовать зависимости. Такая архитектура предполагает наличие вызовов функций только к концентратору событий или к методам, локальным для объекта или модуля.

Давайте взглянем на несколько классических вариантов реализации входа в систему. Ниже представлен стандартный вход в систему, использующий Ajax и платформу YUI:

```
YUI().add('login', function(Y) {
 Y.one('#submitButton').on('click', logIn);
  function logIn(e) {
   var username = Y.one('#username').get('value')
          , password = Y.one('#password').get('value')
          , cfg = {
            data: JSON.stringify(
                    {username: username, password: password })
          , method: 'POST'
          , on: {
                  complete: function(tid, resp, args) {
                    if (resp.status === 200) {
                      var response = JSON.parse(resp.responseText);
                      if (response.loginOk) {
                              userLoggedIn(username);
                      } else {
                              failedLogin(username);
                      }
                    } else {
                              networkError(resp);
                    }
            }
          }
          , request = Y.io('/login', cfg)
    ;
        }
 }, '1.0', { requires: [ 'node', 'io-base' ] });
```

27 строк кода, обрабатывающего вход в систему, — не очень плохо. Но давайте рассмотрим эквивалентное решение на основе событий-

ного программирования (мы будем использовать модуль YUI Event Hub):

```
YUI().add('login', function(Y) {
 Y.one('#submitButton').on('click', logIn);
 function logIn(e) {
  var username = Y.one('#username').get('value')
   , password = Y.one('#password').get('value')
  ;
  eventHub.fire('logIn'
   , { username: username, password: password }
   , function(err, resp) {
    if (!err) {
     if (resp.loginOk) {
      userLoggedIn(username);
     } else {
      failedLogin(username);
     }
     } else {
      networkError(err);
     }
   });
 }
}, '1.0', { requires: [ 'node', 'EventHub' ] });
```

На этот раз код занял всего 18 строк, экономия кода составила 33%, что уже очень и очень неплохо. Но в дополнение к меньшему числу строк кода существенно упрощено его тестирование. Чтобы проиллюстрировать это, давайте проведем модульное тестирование. Детально модульное тестирование будет рассмотрено нами в главе 4, поэтому особо не обращайте внимание на синтаксис, если не знакомы с YUI Test. Начнем с Ajax-версии:

```
YUI().use('test', 'console', 'node-event-simulate'
 , 'login', function(Y) {
 // Фабрика для «подделки» Y.io
 var getFakeIO = function(args) {
  return function(url, config) {
        Y.Assert.areEqual(url, args.url);
        Y.Assert.areEqual(config.data, args.data);
        Y.Assert.areEqual(config.method, args.method);
        Y.Assert.isFunction(config.on.complete);
```

```
            config.on.complete(1, args.responseArg);
        };
    }
    , realIO = Y.io;
```

Это стандартный код YUI, загружающий внешние модули, включая наш модуль `login`, который мы и будем тестировать. Код также содержит фабрику для создания поддельной инстанции `Y.io`, которая является YUI-оболочкой вокруг `XMLHttpRequest`. Этот объект гарантирует, что ему будут переданы корректные значения. Наконец, мы сохраняем ссылку на реальный экземпляр `Y.io` так, чтобы его можно было восстановить для других тестов. Сам код теста выглядит так:

```
var testCase = new Y.Test.Case({
  name: "test ajax login"
  , tearDown: function() {
        Y.io = realIO;
  }
  , testLogin : function () {
    var username = 'mark'
      , password = 'rox'
  ;
  Y.io = getFakeIO({
    url: '/login'
    , data: JSON.stringify({
        username: username
        , password: password
    })
    , method: 'POST'
    , responseArg: {
      status: 200
      , responseText: JSON.stringify({ loginOk: true })
    }
  });
  userLoggedIn = function(user) {
        Y.Assert.areEqual(user, username); };
  failedLogin = function() {
        Y.Assert.fail('Вход в систему успешен!'); };
  networkError = function() {
        Y.Assert.fail('Вход в систему успешен!'); };
  Y.one('#username').set('value', username);
```

```
        Y.one('#password').set('value', password);
        Y.one('#submitButton').simulate('click');
      }
    });
```

Вышеприведенный код тестирует успешную попытку входа. Мы настраиваем наш поддельный объект `Y.io` и тестируем различные функции входа. Затем мы настраиваем HTML-элементы: заполняем их известными значениями, которые могут быть переданы при нажатии кнопки **Submit (Отправить)**. Отметим функцию **teardown**, позволяющую восстановить объект `Y.io` в его исходное значение. Имитация `Y.io` немного неуклюжая, поскольку Ajax имеет дело только со строками, и мы должны контролировать HTTP-статус от сервера. Для полноты картины рассмотрим остальную часть нашего теста:

```
    var suite = new Y.Test.Suite('login');
    suite.add(testCase);
    Y.Test.Runner.add(suite);
    // Инициализируем консоль
    new Y.Console({
            newestOnTop: false
    }).render('#log');
    Y.Test.Runner.run();
  });
```

Теперь давайте посмотрим на модульный тест для событийно-ориентированного кода. Начальный фрагмент теста выглядит так:

```
YUI().use('test', 'console', 'node-event-simulate'
, 'login', function(Y) {
// Фабрика для имитации EH
var getFakeEH = function(args) {
return {
  fire: function(event, eventArgs, cb) {
    Y.Assert.areEqual(event, args.event);
    Y.Assert.areEqual(JSON.stringify(eventArgs),
                      JSON.stringify(args.data));
    Y.Assert.isFunction(cb);
    cb(args.err, args.responseArg);
  }
 };
};
});
```

Здесь мы видим тот же шаблон и подобную фабрику для имитации обработчика событий. Несмотря на это, данный код более чист и прост, небольшая сложность возникает при сравнении двух объектов на равенство, поскольку YUI не предоставляет стандартного метода сравнения объектов.

Теперь рассмотрим фактический код теста:

```
var testCase = new Y.Test.Case({
name: "test eh login"
, testLogin : function () {
var username = 'mark'
, password = 'rox'
;
eventHub = getFakeEH({
event: 'logIn'
, data: {
username: username
, password: password
}
, responseArg: { loginOk: true }
});
userLoggedIn = function(user) {
Y.Assert.areEqual(user, username); };
failedLogin = function() {
Y.Assert.fail('login should have succeeded!'); };
networkError = function() {
Y.Assert.fail('login should have succeeded!'); };
        Y.one('#username').set('value', username);
        Y.one('#password').set('value', password);
        Y.one('#submitButton').simulate('click');
  }
});
```

Данный код по своей сути очень похож на рассмотренный нами ранее для Ajax-версии. Мы получаем сымитированную версию концентратора событий, заполняем HTML-элементы и нажимаем кнопку **Submit** для имитации входа в систему.

Однако этот пример иллюстрирует, что в случае с событийно-ориентированным программированием размер кода (как основного, так и кода тестирования) существенно меньше. Например,

событийно-ориентированный код обрабатывает междоменную (безопасную или нет) коммуникацию без всяких изменений. У нас нет никакой нужды в JSONP, Flash, скрытых фреймах (iframe) или любых других приемах, добавляющих сложность и коду, и к тестам. Сериализация и десериализация обрабатываются прозрачно. Ошибки передаются как объекты, а не как коды состояний HTTP. Конечные точки абстрагируются из URL в произвольные строки. И наконец (и что, возможно, наиболее важно), событийно-ориентированная архитектура полностью освобождает нас от тирании инстанцирования и поддержания сильно связанных указателей на внешние объекты.

Код теста для Ajax-запросов значительно увеличивается при использовании каких-то дополнительных фич `XMLHttpRequest` и `Y.io` (например, таких как обратные вызовы `success` и `failure`), которые также нуждаются в имитации.

Ответы на порожденные события

Есть три возможных варианта реакции на порожденное событие: нет ответа, универсальный ответ, определенный ответ.

Самый простой способ — когда вообще не нужен никакой ответ. В типичной системе событий отправка события очень проста, но получение ответов более проблематично.

Вместо занудных рассуждений об особенностях того или иного способа реакции на событие, давайте рассмотрим несколько примеров. Начнем с самого простого случая, когда *ответ не нужен*. В этом случае мы просто порождаем событие и двигаемся дальше:

```
eventHub.fire('LOG', { level: 'debug', message: 'This is cool' });
```

Если какой-то специфический ответ на порожденное событие не нужен и вас устроит *универсальный ответ*, удаленный слушатель может породить универсальное событие в ответ. Например, в случае регистрации нового пользователя может быть порождено универсальное событие `USER_REGISTERED` и этого будет более чем достаточно:

```
// где-то на сервере
eventHub.on('REGISTER_USER', function(obj) {
  // заносим obj.name в БД пользователей
  eventHub.fire('USER_REGISTERED', { name: obj.name });
});
// где-то в глобальной области
eventHub.on('USER_REGISTERED', function(user) {
  console.log(user.name + ' зарегистрирован!');
});
// где-то в определенном месте (порождаем событие)
eventHub.fire('REGISTER_USER', { name: 'mark' });
```

В этом примере обработчик события `REGISTER_USER` находится где-то на сервере, его задача — записать имя пользователя в базу данных. После регистрации пользователя путем его добавления в базу данных обработчик порождает другое событие – универсальный ответ `USER_REGISTERED`. Другой слушатель ждет данное событие. Что будет делать этот обработчик события, зависит от логики приложения, например, он может обновить интерфейс пользователя. Конечно, у вас может быть несколько слушателей для каждого события, например, один обработчик события `USER_REGISTERED` может обновлять панель управления, другой может обновлять интерфейс конкретного пользователя, а третий — отправлять администраторам электронное сообщение с информацией о пользователе.

Наконец, в некоторых случаях нужен *определенный ответ*, отправляемый непосредственно создателю события. Если нужен прямой ответ, концентратор событий предоставляет обратные вызовы (callback) для создателя события. Callback-функция передается в качестве параметра создателем события в концентратор событий, например:

```
// где-то на сервере
eventHub.on('ADD_TO_CART'
        , function(userId, itemId, callback) {
                d.cart.push(itemId);
                callback(null, { items: userId.cart.length });
});
// Тем временем, где-то в другом месте (возможно, в браузере)
function addItemToCart(userId, itemId) {
        eventHub.fire('ADD_TO_CART'
          , { user_id: userId, item_id: itemId }
```

```
        , function(err, result) {
          if (!err) {
                console.log('Cart now has: ' + result.cart.items + ' items');
          }
        });
}
```

Здесь мы порождаем универсальное событие `ADD_TO_CART` с некоторой информацией и ждем обратного вызова. Callback-функция, переданная концентратору событий, вызывается с данными, предоставленными слушателем.

3.3. Контекст применения событийно-ориентированной архитектуры

Событийно-ориентированная архитектура и MVC-методы

Можно ли сравнить событийно-ориентированную архитектуру с архитектурой MVC (Model-View-Controller, Модель-представление-контроллер)? Эти две архитектуры довольно похожи, но все-таки у них есть отличия.

Самое большое отличие между ними состоит в том, что в событийно-ориентированной архитектуре модели как бы устранены и неявны. Например, при использовании MVC класс `Model` инстанциируется для каждой строки базы данных, этот класс предоставляет данные и методы, которые оперируют этими данными.

В событийно-ориентированной архитектуре модели — это просто хэши, хранящие только данные; эти данные передаются с использованием событий для слушателей (при желании вы можете называть их моделями), чтобы работать с ними. Это разделение данных и функциональности по идее обеспечивает отличную тестируемость, экономию памяти и более высокую масштабируемость, и это все еще с преимуществами соединения данных и методов. Вместо множества отдельных созданных объектов `Model` есть много хэшей данных и один объект для работы с ними.

Далее, контроллер из MVC – в событийно-ориентированной архитектуре это просто концентратор событий, который вслепую передает события между представлениями и моделями. У концентратора события может быть некоторая логика, например, он может передавать события конкретным слушателям вместо того, чтобы передавать их всем слушателям (подобно тому как работает коммутатор (switch) в сети Ethernet). Концентратор может также обнаружить события без слушателей и возвратить сообщение об ошибке.

Представления получают данные, переданные им через события вместо опрашивания моделей. На инициированное пользователем событие представления "моделируют" данные и порождают надлежащее событие. Так, все необходимые обновления пользовательского интерфейса производятся через уведомления с помощью событий.

Событийно-ориентированная архитектура и объектно-ориентированное программирование

Основной принцип объектно-ориентированного программирования — хранить данные вместе с методами для их обработки. Другой принцип объектно-ориентированного программирования — повторное использование, то есть более общие суперклассы могут быть снова использованы другими подклассами. Также ООП предоставляет другую форму связи — наследование. Что же касается событийно-ориентированного программирования, то никто не мешает программисту использовать принципы объектно-ориентированного программирования. Однако есть парочка нюансов.

Для начала все приложение не будет объектно-ориентированным. Кроме того, нет никаких цепочек объектов, соединенных только наследованием или интерфейсным связыванием, а само приложение не работает в одном единственном потоке или процессе, а использует несколько потоков. Но самое большое различие состоит в том, что данные при событийно-ориентированном подходе не хранятся в объектах. Объекты «Одиночка» (Singleton) инстанциируются с методами, которые обрабатывают данные, а сами данные передаются в эти объекты для их обработки.

Нет никаких «открытых» (public), «закрытых» (private) или «защищенных» (protected) модификаторов — все данные закрытые. Един-

ственная возможная связь с "внешним" миром — через событийно-ориентированный API. Не нужно волноваться, что внешний объект смешает внутренние данные вашего объекта. Нет проблем жизненного цикла — объекты живут на протяжении всей жизни приложения, поэтому конструкторы (если нужны) вызываются один раз — в самом начале, а деструкторы и вовсе не нужны.

Событийно-ориентированная архитектура и SaaS

Событийно-ориентированная архитектура существенно упрощает реализацию подхода SaaS (Software as a Service, программное обеспечение как услуга). Каждая независимая часть проекта может присоединиться к концентратору событий, чтобы предоставить услугу безотносительно всех других сервисов, используемых в приложении.

Все это вы можете для простоты представить в виде репозитария сервисов, в который вы можете либо загрузить свое приложение, либо сделать его доступным удаленно через Интернет. При этом ваше приложение-сервис должно соединиться с их концентратором событий, чтобы получить доступ к их сервисам. "API" для всех сервисов — уникальное имя события, с которым должно быть связано действие для обработки в рамках вашего сервиса/приложения и с которым могут быть связаны определенные данные. Сервис также может запустить другое событие или использовать механизм обратного вызова для ответов, если в этом есть необходимость.

3.4. Веб-ориентированные приложения

Большинство веб-приложений тесно взаимодействуют с веб-сервером. Это не очень удачное решение, поскольку, как мы уже упоминали ранее, изначально у HTTP-серверов никогда не было той функциональности, которая требуется от них сегодня. HTTP-серверы построены для работы сугубо по протоколу передачи гипертекста (HTTP), то есть для обслуживания статического контента. Все остальное было "прикручено" в виде дополнительных модулей, что в результате привело к излишне сложным веб-приложениям.

Смешивание логики приложения с веб-сервером делают веб-сервер центральным концентратором и жестко привязывают ваше приложение к HTTP и вообще к самому веб-серверу. Для организации двухсторонней асинхронной передачи средствами HTTP было создано множество трюков, «костылей» и проч., но определенные сложности все же еще есть. В то же время библиотека `socket.io` не только обеспечивает двусторонний обмен данными по HTTP, но и делает поддержку веб-сокетов абсолютно прозрачной.

Веб-приложения также страдают от проблемы «разного происхождения», когда фрагменты кода от разных источников, собираемые воедино на стороне браузера или на стороне сервера, не могут корректно или в полной мере друг с другом взаимодействовать. В HTML5 появился метод `postMessage`, использующий передачи сообщений между такими фрагментами, но при использовании `socket.io` можно обойтись без его использования.

Концентратор событий в событийно-ориентированной архитектуре является именно концентратором, а не веб-сервером. Это предпочтительно, поскольку именно в этом состоит основная задача при построении архитектуры веб-ориентированных приложений/сервисов на основании множественных источников, фрагментов кода, сервисов и проч. (что мы, по сути, и имеем в Интернете), а функциональность веб-сервера сосредотачивается в одном из модулей системы (который так и называется) и предоставляется по требованию наравне с сервисами других модулей. При этом концентратор событий построен с нуля, не является «костыльным» решением, а потому предельно ориентирован на эффективную реализацию своего предназначения.

3.5. Тестирование событийно-ориентированной архитектуры

Тестирование событийно-ориентированной архитектуры заключается в простом вызове функции, реализующей действие. Вот интересный случай:

```
// Некоторый дескриптор БД
function DB(eventHub, dbHandle) {
  // Функция добавления пользователя
```

```
    eventHub.on('CREATE_USER', addUser);
    function addUser(user) {
      var result = dbHandle.addRow('user', user);
      eventHub.fire('USER_CREATED',
        {
          success: result.success
          , message: result.message
          , user: user
        }
      );
    }
}
```

В этом коде есть несколько любопытных вещей, на которые нужно обратить ваше внимание. Во-первых, здесь нет никаких обратных вызовов, а вместо них используются события для широковещательного оповещения об успешном или неудачном завершении этой операции.

Возможно, вы захотите использовать обратный вызов, чтобы только вызывающая сторона знала, как (успешно или нет) завершилась эта операция, поскольку при передаче результата выполнения этой операции широковещательно другие заинтересованные стороны тоже смогут узнать, успешно ли создан пользователь или нет.

С другой стороны, широковещательный режим может быть очень полезен, поскольку таким образом можно проинформировать множество разных клиентов о том, создан ли новый пользователь или нет. В этом случае любой браузер или клиент стороны сервера будут уведомлены относительно этого события (то есть о создании нового пользователя). Также это позволит отдельному модулю обработать результаты события создания пользователя вместо того, чтобы заключать код обработки результата операции в функцию `addUser` (то есть можно установить обработчик события создания пользователя вместо того, чтобы добавлять код в функцию `addUser`).

Конечно, код, получающий уведомление о создании пользователя, может быть в одном файле с функцией `addUser`, но обычно в этом нет смысла. На практике уведомление о событии получает модуль на стороне сервера или клиента, находящийся в отдельном файле, — таким модулям нужно знать, создался ли пользователь или нет, и

принять соответствующие меры. Например, слушатель на стороне клиента может обновить интерфейс пользователя, а слушатель на стороне сервера может обновить какие-то внутренние структуры.

Здесь `eventHub` и `databaseHandle` введены в конструктор, что помогает тестированию и принципу модульности. При этом мы не создаем объекты, мы только регистрируем слушателей событий во введенном `eventHub`. Это и есть сущность событийно-ориентированной архитектуры: регистрируем слушателей событий и не инстанциируем объекты.

Браузеро-ориентируемый обработчик события может выглядеть так:

```
eventHub.on('USER_CREATED', function(data) {
        dialog.show('User created: ' + data.success);
});
```

Обработчик стороны сервера может быть примерно таким:

```
eventHub.on('USER_CREATED', function(data) {
        console.log('User created: ' + data.success);
});
```

Предыдущий код откроет диалоговое окно при создании пользователя (не имеет разницы, где было инициировано создание пользователя — в этом браузере, в каком-то другом браузере или даже на стороне сервера). Если это то, чего вы пытаетесь достичь, используйте широковещательные события, если нет — используйте обратный вызов. Также возможно породить событие, как на стороне клиента, так и сервера. Код на стороне клиента может выглядеть так:

```
eventHub.fire('CREATE_USER', user);
```

На стороне сервера:

```
eventHub.fire('CREATE_USER', user);
```

Проще уже некуда!

Для тестирования этой функции вы можете использовать имитацию концентратора событий и заглушенный обработчик базы данных, что позволит проверить, правильно ли порождаются события:

```
YUI().use('test', function(Y) {
```

```
var eventHub = Y.Mock()
  , addUserTests = new Y.Test.Case({
    name: 'add user'
    , addOne: function() {
      var user = { user_id: 'mark' }
        , dbHandle = { // заглушка БД
          addRow: function(user) {
            return { user: user
                  , success: true
                  , message: 'ok' };
            }
        }
      ;
      DB(eventHub, dbHandle); // тестовые версии
      Y.Mock.expect(eventHub, 'fire',
          [
            'USER_CREATED'
            , { success: true: message: 'ok', user: user }
          ]
      );
      addUser(user);
      Y.Mock.verify(eventHub);
    });
  Y.Test.Runner.add(addUserTests);
  Y.Test.Runner.run();
});
```

В этом случае для тестирования функции `addUser` мы используем оба приема тестирования: и использование имитаций – mock (для концентратора событий), и использование заглушек – stub (для дескриптора БД). Событие `fire`, как ожидается, вызовет на объекте `eventHub` функцию `addUser` с соответствующими аргументами. Функция `addRow` объекта-заглушки `DB` просто сообщает, что создание пользователя прошло успешно. Оба данных объекта (концентратор событий и дескриптор БД) вводятся в объект `DB` для тестирования.

В результате всего этого простого тестирования мы можем убедиться, что функция `addUser` генерирует надлежащее событие с надлежащими аргументами.

Сравните данное тестирование со стандартным подходом, когда используется инстанциированный объект `DB` с его прототипом функции `addUser`:

```
var DB = function(dbHandle) {
        this.handle = dbHandle;
};
DB.prototype.addUser = function(user) {
        var result = dbHandle.addRow('user', user);
        return {
                success: result.success
                , message: result.message
                , user: user
        };
};
```

Как ко всему этому получить доступ со стороны клиента? А вот:

```
transporter.sendMessage('addUser', user);
```

Глобальный или централизованный механизм передачи сообщений (в данном случае мы получаем к нему доступ посредством объекта transporter) отправит сообщение обратно серверу, используя Ajax или что-то подобное (способ передачи сообщения нас сейчас мало волнует, но на практике обычно используется технология Ajax, позволяющая сценариям JS отправлять информацию на сервер без перезагрузки страницы). Отправка сообщения о создании пользователя происходит, как правило, после создания или обновления объекта модели user на стороне клиента. Части, которые должны отслеживать запросы и ответы, а также код стороны сервера нуждаются в маршруте, по которому можно отправить сообщение глобальному объекту `DB`, обработать ответ и отправить ответ обратно клиенту.

Далее приведен тест для этого кода:

```
YUI().use('test', function(Y) {
  var addUserTests = new Y.Test.Case({
    name: 'add user'
    , addOne: function() {
      var user = { user_id: 'mark' }
        , dbHandle = { // заглушка БД
          addRow: function(user) {
            return {
```

```
                user: user
                , success: true
                , message: 'ok'
               );
          }
        }
        , result
        ;
    DB(dbHandle); // Тестовые версии
    result = addUser(user);
    Y.Assert.areSame(result.user, user.user);
    Y.Assert.isTrue(result.success);
    Y.Assert.areSame(result.message, 'ok');
   }
  })
  ;
  Y.Test.Runner.add(addUserTests);
  Y.Test.Runner.run();
});
```

Чтобы клиент мог добавить нового пользователя, нужно создать новый протокол между клиентом и сервером, также нужно создать маршрут в сервере, чтобы передать сообщение «add user» объекту DB, после чего результаты добавления пользователя должны быть сериализованы и переданы обратно вызывающей стороне. Нужно проделать огромную работу, чтобы создать подобную инфраструктуру для каждого типа сообщений. В конечном счете вы придете к необходимости реализовать мощную систему RPC (Remote Procedure Call (RPC)) — класс технологий, позволяющих компьютерным программам вызывать функции или процедуры в другом адресном пространстве), которую концентратор событий предоставляет вам просто так.

Это яркий пример тех самых 85% шаблонного кода, который вы не должны реализовывать самостоятельно. Все приложения по сути сводятся к передаче сообщений и управлению заданиями. Приложения передают сообщения и ожидают ответы на них. Написание и тестирование шаблонного кода добавляет много лишних издержек любому приложению. Что, если вы также хотите добавить интерфейс командной строки вашему приложению? Это уже другой путь выполнения кода, который вы также должны реализовать и об-

служивать. Возложите передачу сообщений на плечи концентратора событий, не изобретайте велосипед заново!

3.6. Недостатки и "узкие" места событийно-ориентированной архитектуры

Событийно-ориентированная архитектура – далеко не идеальное решение, и за рядом преимуществ имеется целый ряд недостатков. Причем необходимо понимать, что одно и то же качество в одних условиях может восприниматься как достоинство, а в других – как недостаток. Сейчас мы рассмотрим некоторые «узкие места», которые нужно иметь в виду при использовании событийно-ориентированной архитектуры.

Масштабируемость

Концентратор событий создает одну большую точку отказа: если концентратор выйдет из строя, ваше приложение перестанет работать.

Вам нужно или использовать ряд концентраторов событий вкупе с какой-то системой балансировки нагрузки, или же предусмотреть резервный концентратор событий, который будет использоваться в случае отказа основного концентратора.

Широковещание

Много событий, передаваемых широковещательно, потенциально может привести к большому количеству трафика. Нужно реализовать защиту, гарантирующую, что события, предназначенные для локального использования, не будут отправлены на концентратор.

Также сам концентратор может действовать как коммутатор (switch), если он знает, какой клиент и какое событие прослушивает: тогда он будет отправлять определенное событие только определенному клиенту, вместо того, чтобы отправлять его всем клиентам, подключенным к концентратору. О коммутаторах событий мы поговорим чуть ниже.

Проверка времени выполнения

Орфографическая ошибка в имени функции или метода приведет к ошибке выполнения. Компилятор не может проверить строковые имена событий на орфографические ошибки. Поэтому настоятельно рекомендуется использовать перечисления (enumeration) или хэши для имен событий вместо того, чтобы указывать имена событий много раз вручную. Рассмотрим прекрасный способ проверки ваших имен событий времени выполнения:

```
// Возвращает объект с функциями 'on' и 'fire' для указанного имени события
hub.addEvent = function(eventName) {
  var _this = this;
  this.events[eventName] = {
    on: function(callback) {
                _this.on.call(_this, eventName, callback);
    }
    , fire: function() {
                Array.prototype.unshift.call(arguments, eventName);
                this.fire.apply(_this, arguments);
    }
  };
  return this.events[eventName];
};
```

Этот код можно использовать примерно так:

```
var clickEvent = hub.addEvent('click');
clickEvent.on(function(data) { /* получили событие click! */ });
clickEvent.fire({ button: 'clicked' }); // сгенерировали событие click!
```

Теперь у вас есть проверка имен событий времени выполнения.

Безопасность

Если вы боитесь, что кто-то подключится к вашему концентратору событий и вставит свои чужеродные события, нужно использовать реализацию концентратора событий с аутентификацией клиентов. В этом случае подключиться к концентратору событий смогут только «доверенные» клиенты, прошедшие аутентификацию. Если вы хотите зашифровать соединения с концентратором событий, вы можете написать свой собственный концентратор события, используя библиотеку `socket.io`, поддерживающую зашифрованные соединения.

Состояние сессии

Сейчас мы поговорим о том, как происходит сохранение состояния сеанса. Состояние сеанса, которое обычно предоставляется веб-сервером посредством Cookie, пересылается от веб-сервера к модулю. После загрузки веб-приложения может быть создан новый объект сеанса и сохранен полностью независимо от веб-сервера. JavaScript, выполняющийся на стороне клиента, запишет маркер сессии как Cookie, после чего маркер сеанса станет доступен и веб-серверу. Концентратор событий вставляет сеанс в события, делая невозможным подмену одного сеанса другим.

Доверенный модуль (обычно находится на стороне сервера) получает ключ сеанса, поэтому состояние события передается в качестве параметра или поле хеша.

Другими словами, состояние сеанса хранится и передается посредством Cookies. Одна сторона (клиент) записывает состояние сеанса в Cookie, после чего вторая сторона (сервер) может прочитать состояние сеанса и использовать его в своих целях. Благодаря концентратору событий становится невозможной подмена одного сеанса другим.

3.7. Интеллектуальный концентратор: коммутатор событий

Коммутатор (переключатель) событий улучшает событийно-ориентированную архитектуру, делая ее более модульной и более масштабируемой и развертываемой. Если добавить немного логики в концентратор событий и в сами события, вы упростите развертывание и управление своими приложениями.

Разделив еще ваши события на две группы: широковещательные и одноадресные, вы максимально реализуете потенциал использования коммутатора вместо концентратора. Главная выгода такого решения (помимо экономии пропускной полосы канала) — отказоустойчивое развертывание.

Одно из самых больших ожиданий событийно-ориентированной архитектуры — модульность. Типичное монолитное приложение

заменяется многими независимыми модулями, что является очень значимым для тестируемости и сопровождения, а использование коммутатора вместо концентратора играет огромную роль для развертывания.

Коммутатор событий знает разницу между одноадресными событиями и широковещательной передачей. Широковещательные события ведут себя как и все события при использовании концентратора событий. Одноадресные события отправляются только определенному слушателю, который зарегистрирован для прослушки этого события.

Одноадресные передачи существенно упрощают развертывание, что и будет показано в следующем разделе.

Развертывание

У монолитного web-приложения обычно вся логика находится на стороне сервера и тесно переплетена с HTTP-сервером. Развертывание новой версии такого приложения влечет за собой пересборку всей логики всего приложения и еще вдобавок перезапуск веб-сервера. И это при каждом обновлении.

В противоположность этому логика событийно-ориентированного приложения совсем не связана с веб-сервером и фактически распределена на множестве независимых модулей.

С использованием коммутатора событий существенно упрощается развертывание новой версии кода, поскольку отдельные модули можно развернуть независимо друг от друга и независимо от веб-сервера. Несмотря на то, что у приложения появилось много "подвижных" частей, с которыми приходится работать, вы получаете прекрасный контроль над развертыванием своего кода и его сопровождением.

В ходе развертывания, при временном отключении какого-либо модуля (слушателя), обрабатываемое им событие не теряется, так как на это существует коммутатор событий. Далее мы подробно рассмотрим, как он действует в подобных ситуациях применительно к одноадресным событиям и широковещательным.

Одноадресные события

Одноадресные события вроде `depositMoney` в нижеприведенном коде должны иметь только одного слушателя. В этом случае слушатель должен уведомить коммутатор событий, что это событие является одноадресным, и что только он должен его прослушивать.

Когда мы будем готовы к обновлению/замене какого-либо модуля, нам нужно запустить новую версию модуля, который объявит коммутатору события, что теперь он будет единственным слушателем определенного события. Коммутатор уведомит более старый модуль, что тот будет заменен и теперь может закрыться после обработки текущих событий. В результате не будут потеряны события, будет закрыт старый модуль и введен в рабочее состояние новый модуль без перерывов в работе приложения:

```
eventSwitch.on('depositMoney', function(data) {
        cash += data.depositAmount;
        eventSwitch.emit('depositedMoney', cash);
}, { type: 'unicast' });
```

Когда мы начнем прослушивать событие `depositMoney`, данный код информирует коммутатор событий, что данное событие `depositMoney` является одноадресным и что все события с этим именем должны быть отправлены только этому слушателю. Все, что мы должны сделать, — это добавить третий параметр методу `on` с некоторыми метаданными о событии, который клиент будет передавать коммутатору.

Тем временем коммутатор заменит текущего слушателя для этого события новым слушателем и отправит событие «кто-то заменил вас» предыдущему слушателю (если таковой был установлен). Любой слушатель должен также прослушивать это событие:

```
eventHub.on('eventClient:done', function(event) {
        console.log('DONE LISTENING FOR ' + event);
        // заканчиваем обработку событий:
        process.exit(0);
});
```

Событие `eventClient:done` информирует, что этот слушатель больше не будет получать события и что как только он обработает оставшиеся события, он может безопасно завершить работу.

Рис. 3.3. Коммутатор событий для непрерывной обработки событий

Коммутатор событий гарантирует, что никакие события не будут отброшены и что ни одно из них не будет отправлено старому слушателю. Как говорится, лучше один раз увидеть, чем сто раз услышать, поэтому посмотрите на рис. 3.3. На этом рисунке время идет сверху вниз. Модуль слева регистрируется для прослушки одноадресного события, и события стали поступать к нему. Чуть позже регистрируется новый модуль (справа) для прослушки того же одноадресного события. В результате этого первый модуль получает событие `done`, теперь все новые запросы будут отправлены модулю справа, в то время как модуль слева закончит обработку полученных событий и закончит работу. Новый модуль продолжит обработку запросов событий, пока какой-либо еще другой модуль не уведомит коммутатор, что теперь он должен быть ответственным за обработку этого одноадресного события.

Широковещательные события

Безопасное закрытие широковещательных слушателей происходит аналогичным образом. Однако слушатель события самостоятельно передает широковещательное событие `eventClient:done`, чтобы сообщить остальным слушателям указанного события, что они должны закрыться. У коммутатора событий же достаточно «мозгов», чтобы передать это событие всем слушателям, кроме того, кто его отправил. Слушатель, получивший данное событие, больше не будет получать определенные события. После обработки оставшихся событий слушатели могут безопасно завершить работу:

```
eventHub.on('eventClient:done', function(event) {
        console.log('Мне больше не нужно прослушивать событие: '
```

```
            + event);
        eventHub.removeAllListeners(event);
});
```

Обратите внимание на то, что коммутатор событий достаточно сообразителен, чтобы не позволить всем подряд подсоединенным клиентам отправлять события `eventClient:done`, и отбрасывает их (события) в случае получения. Таким образом, в случае чего, клиент злоумышленника не может узурпировать полномочия коммутатора события.

Реализация

EventHub (https://github.com/zzo/EventHub) — простая реализация коммутатора событий, имеющая клиенты для Node.js, YUI3, jQuery. Основан EventHub на библиотеке `socket.io`, может быть установлен при помощи npm:

```
% npm install EventHub
```

Построенная на базе `socket.io`, данная реализация представляет собой централизованный концентратор, работающий под Node.js и предоставляющий функции обратного вызова для прямых ответов. Большинство событий не требуют прямых ответов, поэтому не перестарайтесь с использованием этой функции.

После установки `EventHub` запустить коммутатор можно так:

```
% npm start EventHub
```

В результате будет запущен коммутатор событий, по умолчанию прослушивающий порт 5883. Всех клиентов нужно настроить на использование этого порта и компьютера, на котором установлен коммутатор. Пример настройки клиента Node.js:

```
var EventHub = require('EventHub/clients/server/eventClient.js');
        , eventHub = EventHub.getClientHub('http://localhost:5883');
eventHub.on('ADD_USER', function(user) {
        // Добавляем логику пользователя
        eventHub.fire('ADD_USER_DONE', { success: true, user: user });
});
```

Эти строки инстанциируют клиент коммутатора события, соединяющийся с ранее запущенным коммутатором события (коммутатор

работает на той же машине, что и клиент, но, конечно же, может работать и на другом узле). Просто запустите данный файл:

```
% node client.js
```

Теперь любое событие ADD_USER, сгенерированное любым клиентом, будет перенаправлено этой функции. Рассмотрим клиент, выполняющийся в браузере с использованием YUI3:

```
<script src="http://yui.yahooapis.com/3.18.1/build/yui/yui-min.js">
  </script>
<script src="/socket.io/socket.io.js"></script>
<script src="/clients/browser/yui3.js"></script>
<script>
        YUI().use('node', 'EventHub', function(Y) {
         var hub = new Y.EventHub(io, 'http://myhost:5883');
        hub.on('eventHubReady', function() {
                hub.on('ADD_USER_DONE', function(data) { });
                ... спустя некоторое время ...
                 hub.fire('ADD_USER', user);
                  });
        });
</script>
```

После загрузки YUI3, библиотеки socket.io и YUI3-клиента EventHub будет инстанциирован новый экземпляр коммутатора EventHub. Когда все будет готово, мы начинаем прослушивать события и порождаем событие ADD_USER. Все просто.

Далее приведен тот же пример с использованием jQuery:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
  </script>
<script src="/socket.io/socket.io.js"></script>
<script src="/clients/browser/jquery.js"></script>
<script>
        var hub = new $.fn.eventHub(io, 'http://myhost:5883');
        hub.bind('eventHubReady', function() {
        hub.bind('ADD_USER_DONE', function(data) { });
                ... Спустя некоторое время ...
         hub.trigger('ADD_USER', user);
        });
</script>
```

Процесс тот же: загружается jQuery, библиотека `socket.io` и jQuery-клиент `EventHub`. Коммутатор событий `EventHub` представлен как jQuery-плагин и использует jQuery-синтаксис событий `bind` и `trigger`.

Любое событие, отправленное клиентом, может иметь необязательную функцию обратного вызова, которая указывается как последний параметр:

```
hub.fire('CHECK_USER', 'mark',
  function(result) { console.log('exists: ' + result.exists); });
```

Любой ответчик на это событие задается соответствующей функцией как последний параметр в списке аргументов события для передачи произвольных данных:

```
hub.on('CHECK_USER', function(username, callback) {
        callback({ exists: DB.exists(username) });
});
```

Обратный вызов (callback) вызывает удаленную функцию из пространства вызывающей стороны с предоставленными ей данными вызывающего. Данные сериализуются как JSON, поэтому при желании вы можете получить целевой ответ на произошедшее (сгенерированное) событие (в данном случае речь идет о событии 'CHECK_USER').

Вы также можете использовать метод `addEvent` для проверки имен событий во время компиляции с использованием YUI3:

```
var clickEvent = hub.addEvent('click');
clickEvent.on(function(data) { /* получаем событие click! */ });
clickEvent.fire({ button: 'clicked' }); // отправляем событие click!
```

В случае с jQuery вы будете использовать вместо методов `on` и `fire` методы `bind` и `trigger`:

```
var clickEvent = hub.addEvent('click');
clickEvent.bind(function(data) { /*получаем событие click! */ });
clickEvent.trigger({ button: 'clicked' }); // отправляем событие click!
```

Вы также можете использовать функции коммутатора событий для одноадресных событий:

```
hub.on('CHECK_USER', function(username, callback) {
```

```
            callback({ exists: DB.exists(username) });
    }, { type: 'unicast' });
```

Вы можете завершить работу любых других слушателей широковещательных событий:

```
// завершает работу всех слушателей, кроме текущего
hub.emit('eventClient:done', 'user:addUser');
```

Чтобы это все работало, как нужно, вы должны прослушивать следующее событие:

```
hub.on('eventClient:done', function(event) {
console.log('Завершаю слушать событие ' + event);
hub.removeAllListeners(event);
});
```

Данное событие может быть отправлено коммутатором событий, когда стал доступен новый слушатель одноадресного события, или же пользовательским модулем, когда происходит обновление широковещательных модулей.

Сессии (Сеансы)

Доверенные одноадресные слушатели получают данные сессии из порожденных событий. «Доверенный» клиент подсоединяется к коммутатору событий, используя разделяемый пароль:

```
eventHub = require('EventHub/clients/server/eventClient.js')
  .getClientHub('http://localhost:5883?token=secret)
```

Теперь этот клиент может прослушивать одноадресные события. Все одноадресные события получают дополнительные данные сессии в обратных вызовах событий:

```
eventHub.on('user', function(obj) {
        var sessionID = obj['eventHub:session'];


});
```

Ключ `eventHub:session` — уникальная строка (UUID), которая идентифицирует сессию. Доверенные слушатели могут использовать этот ключ при сохранении данных сессии и при доступе к ним. Ключи сессии полностью прозрачны для недоверенных клиентов. И клиенты jQuery, и клиенты YUI3 хранят ключи сессии в Cookie.

При этом HTTP-сервер — всего лишь доверенный клиент коммутатора событий: он получает ключ сеанса из Cookie по HTTP-запросу и решает, какой контент отправить в браузер.

Расширяемость

У библиотеки `socket.io` есть клиенты, написанные на многих языках программирования. Таким образом, используемые вами службы не ограничены только клиентскими реализациями на JavaScript. При необходимости вы можете написать клиент `EventHub` на другом языке программирования, который поддерживает `socket.io`. Это довольно несложно.

Примечание.

Обратите внимание на проект BidSilent (https://github.com/zzo/BidSilent). Он может вам помочь разобраться в том, как работает веб-приложение на основе событийно-ориентированной архитектуры, а также в том, как преобразовать стандартную реализацию веб-сервера в реализацию с концентратором событий.

Глава 4.

Модульное тестирование

Сразу отметим, что когда дело доходит до модульного тестирования, нет никакого волшебного универсального средства, позволяющего нажатием одной кнопки «Тестировать» выявить все проблемы вашего кода. К сожалению.

Чтобы получить результат, разработчику/тестировщику нужно глубоко погрузиться в работу. Утешает то, что результат, скорее всего, будет стоить потраченных усилий: начиная с сохранения вашей зарплаты и заканчивая удобным в сопровождении рабочим кодом.

Сложнее всего написать первый тест. С чего начать? Первым делом нужно ответить себе на несколько важных вопросов. Например, будете ли вы использовать существующую или самодельную тестовую платформу (тестовый фреймворк)? Как автоматизировать процесс сборки? Как собрать, отобразить и отследить тестируемый код? С рассмотрения этих вопросов мы и начнем данную главу.

4.1. Выбор фреймворка (тестовой платформы)

Как нельзя писать хороший JavaScript-код без использования нормального фреймворка, так нельзя и модульное тестирование произвести, пока вы не выберете и не обзаведетесь тестовым фреймворком. Фреймворки предоставляют много разных шаблонов, которые вам не нужно воссоздавать: реализацию асинхронного тестирования, наборы тестов, помощники mock/stub и др.

На сегодняшний день доступно много хороших платформ тестирования, причем с открытым исходным кодом. В этой главе я буду ис-

пользовать фреймворк YUI Test, однако большинство рассмотренных методов будет применимо и к другим платформам, отличаться будет только синтаксис и некоторая семантика.

Прежде чем продолжить, давайте разберемся с терминологией.

Тестовый сценарий (test case)[1], или просто тест, – это набор специализированных команд для автоматизации тестирования определенной части программы. Тестовый набор (test suite) — это набор тестовых сценариев, объединенных по некоторому принципу и предназначенных для проверки определенной части программы.

Лучше всего сгруппировать тесты для определенного модуля в один тестовый набор. Набор может содержать множество тестов, каждый из которых тестирует какой-то определенный аспект модуля. Используя предоставленные набором функции `setUp` и `setDown`, можно легко управлять конфигурацией до и после теста, например сбросом состояния базы данных.

Другая важная часть модульного тестирования — утверждения.

Утверждения — это тесты, проверяющие ожидаемые значения с полученными.

Получили ли вы ожидаемое значение? YUI Test предоставляет полный набор функций утверждения, результаты которых отслеживаются платформой так, чтобы по окончании всех тестов мы могли бы легко увидеть, какие блоки прошли тесты, а какие — нет.

Платформа YUI Test также предоставляет и другую полезную функциональность: асинхронное тестирование, игнорируемые тесты, поддержка имитации (mock) и т.д. Другими словами, YUI Test — полнофункциональное средство для модульного тестирования.

И последнее замечание: чтобы протестировать код с помощью YUI Test, совсем не обязательно изначально разрабатывать этот код средствами фреймворка YUI.

1 Иногда их еще называют "сетевыми случаями".

Загрузить платформу YUI Test можно отсюда: http://yuilibrary.com/projects/yuitest/

Платформа YUI Test отлично подходит для тестирования клиентской части JavaScript, но ее также можно с успехом использовать и для тестирования серверной части. Однако для популярного нынче BDD[2]-тестирования серверной части также доступны две популярных платформы: **Vows** (http://vowsjs.org/) и **Jasmine** (https://github.com/mhevery/jasmine-node).

Напоминаем, что идея BDD состоит в использовании конструкций «повседневного языка» для описания того, что, по вашему мнению, должен делать код вообще или, если «копать глубже», что должны возвращать ваши функции (какой результат должны доставлять). Это все, так сказать, «причуды», а более важным применительно к фреймворкам Vow и Jasmine является то, что они оба поддерживают асинхронное тестирование и могут выполняться через командную строку. Также они обеспечивают очень хорошие способы группировки, выполнения и тестирования как клиентского, так и серверного кода.

4.2. Давайте приступим

Итак, нам нужно произвести модульное тестирование, так давайте приступим к работе. А начнем мы с этого кода:

```
function sum(a, b) {
        return a + b;
}
```

Модульное тестирование подразумевает изоляцию кода и написание тестов только для тестируемого участка кода. Звучит достаточно просто, но на практике это не так. Конечно, с нашей простой функцией `sum` все будет несложно:

```
YUI({ logInclude: { TestRunner: true } }).use('test', 'test-console'
  , function(Y) {
    var testCase = new Y.Test.Case(
      {
                name: 'Sum Test'
```

2 BDD – (behavior-driven development) – разработка, управляемая поведением, или еще переводят как «Разработка, основанная на функционировании».

```
                , testSimple: function () {
                        Y.Assert.areSame(sum(2, 2), 4
                        , '2 + 2 не равно 4?');
            }
        }
    );
// Загружаем его
Y.Test.Runner.add(testCase);
(new Y.Test.Console({
        newestOnTop: false
})).render('#log');
// Запускаем его
Y.Test.Runner.run();
});
```

Это самый простой тест, который только возможен. Первым делом мы создаем новый тестовый сценарий `Y.Test.Case`. При создании теста мы задаем его имя (name) и набор функций, которые мы будем тестировать. Каждая функция выполняется поочередно. Тестовый сценарий загружается в локальный объект `Y.Test.Runner`, и затем выполняются все тесты.

У нас есть код, который нужно протестировать, и мы должны написать сценарий самого теста. Далее мы должны запустить созданные тесты и просмотреть результаты. Если бы мы тестировали только код стороны сервера (например, используя платформу **Vows**), нам был бы нужен только код и соответствующий ему тест. Но поскольку мы тестируем клиентский код JavaScript, нам нужен еще HTML-файл, который свяжет воедино файл с исходным кодом (sum.js) и файл теста (sumTest.js):

```
<html>
    <head>
        <title>Тест Sum</title>
    </head>
    <body>
        <div id='log' class='yui3-skin-sam'></div>
        <script src='http://yui.yahooapis.com/3.18.1/build/yui/yui-min.js'>
         </script>
          <script src='sum.js'></script>
        <script src='sumTests.js'></script>
    </body>
</html>
```

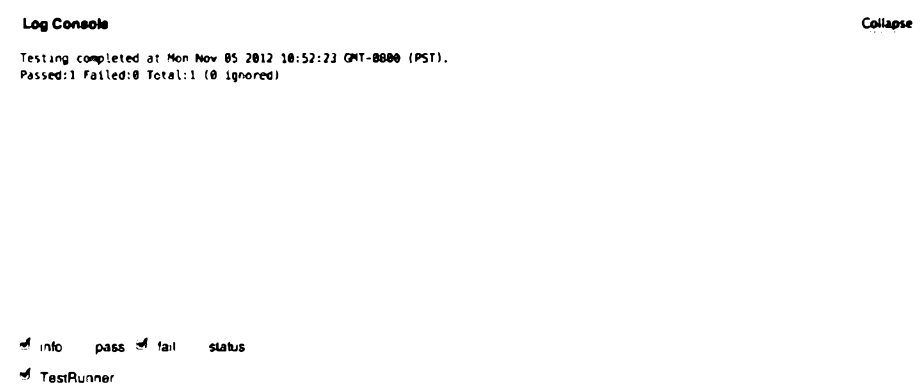


Рис. 4.1. Вывод теста YUI

Вышеприведенный HTML-документ связывает воедино наш YUI3-тест, а именно: код, который нужно протестировать, и код самого теста. Чтобы запустить все тесты и просмотреть их результаты, загрузите этот HTML-документ в браузере. Вы получите вывод, подобный изображенному на рис. 4.1.

Кое-что мы пока пропустим (а именно покрытие кода и выполнение теста в командной строке), но мы обязательно к ним вернемся после того, как узнаем, как написать хорошие тесты.

4.3. Пишем совершенные тесты

К сожалению, мы редко тестируем такие простые функции, как `sum`. У подобных функций обычно нет побочных эффектов, и возвращаемые ими значения зависят только от параметров функции. Однако даже в таких простых функциях могут быть «глюки». Что если вместо чисел будут переданы две строки? Что если параметры функции — строка и целое число? Что если будут переданы значения null, undefined или NaN? Даже самые простые функции требуют тщательного тестирования.

Еще хуже, когда метрика одного модульного тестирования счастливо сообщает от 100%-ом покрытии кода при тестировании всего одного нашего модуля! Очевидно, единственный тест не может быть эффективным, зато менеджер, посмотрев на отчет покрытия кода, может ошибочно прийти к заключению, что раз у этой функ-

ции есть 100%-ое покрытие кода, она полностью протестирована и надежна.

Так как же написать совершенный тест? Есть несколько показателей, на которые нужно ориентироваться. И покрытие кода — это только один из показателей. Другим не менее важным показателем являются граничные и не граничные значения параметров. Учитывая, что зависимости функции будут имитироваться для изоляции тестирования, вам, как тестеру, нужно тщательно протестировать параметры функции. Для примера давайте возьмем уже известную нам функцию `sum`. Данная функция принимает два параметра. Каждый параметр может быть одного из шести типов (число, строка, объект, undefined, null или булевый). Таким образом, совершенный набор модульных тестов тестирует все возможные комбинации этих параметров. Но что он должен протестировать? Что означает сложить null и undefined?

Два самых важных фактора хорошего модульного тестирования — это изоляция и область действия. Как будет показано в следующих подразделах, изоляция и область действия тесно связаны.

Изоляция

Модульному тесту нужно передавать самый минимум кода, только то, что действительно нужно для выполнения теста. Причем чем меньше кода тестируется, тем лучше. Любой дополнительный код может повлиять либо на сам тест, либо на тестируемый код и вызвать дополнительные проблемы. Как правило, при модульном тестировании осуществляется тестирование одного метода (один метод – один тест). Этот метод, вероятно, является частью файла, содержащего класс или модуль. К сожалению, у тестируемого кода часто есть зависимости от других функций и методов, находящиеся в этом файле. Также вдобавок могут быть и внешние зависимости, то есть привязки к другим файлам.

Чтобы избежать загрузки внешних зависимостей, можно использовать имитацию (mock), заглушку (stub) и дублирование теста (test doubles). Все эти стратегии будут рассмотрены позже, а на данный момент важно отметить, что все они пытаются сделать тестируемый код максимально изолированным от других частей кода.

Область действия теста, границы тестирования

Область действия теста — это то, что фактически тестирует ваш тест. Область действия должна быть небольшой. У полностью изолированного метода область действия минимальна.

Как мы уже говорили, идеальные тесты тестируют только один метод и при этом не трогают другие вызываемые данным методом или используемые им методы.

В свою очередь ни один метод не должен пытаться решить больше, чем одну определенную задачу.

4.4. Определение ваших функций

Прежде чем мы углубимся в дебри модульного тестирования, сделаем еще несколько замечаний по существу. С одной стороны, невозможно написать совершенный тест, если вы не знаете точно, что вы будете тестировать и какой, собственно, должен быть результат.

С другой стороны, допустим, что у вас есть тесты. Что определяет, какой должен быть результат у той или иной функции? Тесты? Опираться на тесты для того, чтобы понять, как работает функция, не самый лучший вариант. В обоих случаях решить проблему призвана такая простая вещь, как комментарии, в которых описывается, что функция должна выдавать в качестве результата и т.п.

Таким образом, получается, что для наиболее адекватного понимания кода и эффективной разработки/поддержки нам необходимы блоки комментариев перед функциями, в которых описывается, что должно получиться, а также тесты, которые позволяют проверить корректность работы и функциональность того или иного метода/фрагмента кода.

Блоки комментариев

Возвращаясь к нашему примеру с функцией `sum`, отметим, что она вообще не прокомментирована, и в этом вакууме мы по большому счету не понимаем, что эта функция должна делать. Может, она должна складывать строки? Или объекты? Или числа? Функция

должна сказать нам, что она делает. Поэтому давайте прокомментируем ее, используя синтаксис Javadoc:

```
/*
* Данная функция выполняет сложение двух чисел, в противном случае
* возвращает null
*
* @param a first Первое число
* @param b second Второе число
* @return сумма двух чисел или null
*/
function sum(a, b) {
        return a + b;
}
```

Вот теперь нам понятно, что нужно сделать с функцией: нужно протестировать ее работу с числами. Помните, что невозможно написать хороший тест для неправильно определенной функции. Также невозможно написать хороший тест, когда непонятно, что должна делать функция.

Первое правило на пути к идеальному тесту — у вас должны быть полностью определены спецификации функций.

Комментарии крайне важны, и ошибки в комментариях чрезвычайно опасны и для тестеров, и для специалистов по обслуживанию. К сожалению, нет никакого способа автоматически определить, есть ли ошибки в комментариях, но использование инструментов вроде **jsmeter**, по крайней мере, предупредит вас о наличии функций в вашем коде вообще без комментариев. Наряду с размером кода отсутствие комментариев и ошибки в них являются вторым индикатором того, что код проблемный.

Было бы идеально, чтобы вы (или кто-то еще) смогли написать тесты, просто взглянув на комментарии функции. Эти комментарии должны быть созданы с помощью JSDoc, YUIDoc или Rocco, что сделает комментарии чрезвычайно полезными. Очень важно поддерживать определение функции в актуальном состоянии. Если изменились параметры или возвращаемые значения, убедитесь, что вы также обновили комментарии.

Некоторые разработчики вместо написания комментариев используют техники предварительного тестирования вроде TDD, то есть сначала они пишут тесты, а потом уже код. Даже если вы сначала протестировали модуль, а потом уже написали его код, это не освобождает вас от написания комментариев. Ведь комментарии очень важны для специалистов по обслуживанию, которые будут сопровождать ваш код в будущем. Взглянув на комментарий, они смогут понять, что делает ваша функция. Также комментарии очень важны при повторном использовании кода, когда вы публикуете ваш код, чтобы его могли использовать другие разработчики. Если у вашего кода есть легкодоступная документация, другие разработчики будут использовать именно вашу функцию, а не писать собственную, выполняющую те же действия.

Тесты как первооснова

В принципе, если вы пишете тесты перед кодированием, они могут использоваться в качестве описания того, как должны работать ваши функции. Так как тесты должны поддерживаться, чтобы оставаться в синхронизации с тестируемым кодом (иначе тесты перестанут работать!), тесты (а не комментарии!) более важны при определении того, что делает код.

Теоретически тесты не могут рассинхронизироваться с кодом, в отличие от комментариев, когда разработчик может изменить функцию и забыть должным образом отредактировать комментарий. Однако комментарии во всей этой схеме очень важны, поскольку человеку, который будет поддерживать ваш код, гораздо проще разобраться с комментариями, чем анализировать ваш код и разбираться с тестовыми сценариями, находящимся где-то в другом файле.

4.5. Подходы в тестировании

Позитивное тестирование

Теперь, когда у нас есть надлежащим образом определенная функция, давайте протестируем «корректные» случаи ее вызовов и выполнения. В рамках такого тестирования мы просто передаем разные комбинации двух чисел. Вот операторы `Y.Assert`:

```
Y.Assert.areSame(sum(2, 2), 4, "2 + 2 не равно 4?");
```

```
Y.Assert.areSame(sum(0, 2), 2, "0 + 2 не равно 2?");
Y.Assert.areSame(sum(-2, 2), 0, "-2 + 2 не равно 0?");
Y.Assert.areSame(sum(.1, .4), .5, ".1 + .4 не равно .5?");
```

Вышеприведенный тестовый сценарий не является чем-то из ряда вон выходящим. Он просто тестирует основные случаи, которых будет 99% "в жизни" вашей функции. Такие тесты принято называть позитивными, а само тестирование – позитивным тестированием.

Позитивное тестирование призвано подтвердить основную ожидаемую функциональность тестируемого метода/функции, тестируя основные, типичные случаи использования метода/функции.

Позитивные тесты должны быть написаны и проведены первыми, поскольку они подтверждают основную ожидаемую функциональность тестируемого метода/функции. Здесь вам нужно протестировать все общие случаи того, как функция должна вызываться и использоваться. Если функция не прошла позитивное тестирование, то что-то дальше тестировать бессмысленно.

Негативное тестирование

Негативное тестирование призвано определить наличие и местоположение редких ошибок, когда или функция используется не по назначению или же ей передаются параметры, не входящие в те самые 99% стандартных случаев.

Может ли код обработать неожиданные значения? Что произойдет, если передать функции некорректные или нежелательные параметры? Будут ли они правильно обработаны? Ответы на эти вопросы и позволит дать негативное тестирование.

Рассмотрим пример теста для нашей функции `sum`:

```
Y.Assert.areSame(sum(2),null));
Y.Assert.areSame(sum('1', '2'), null));
Y.Assert.areSame(sum('asd', 'weoi'), null));
```

Данный тест быстро покажет, что наша функция `sum` не ведет себя так, как нужно. Что значит "не так, как нужно"? Ответ находим в

блоке комментариев в описании функции `sum`. А там сказано, что функция должна вернуть null для нечисловых параметров. Без блока комментариев, описывающего поведение этой функции, мы бы и понятия не имели, правильно или нет ведет себя эта функция.

Негативное тестирование обычно не дает то же число выявленных ошибок, что и позитивное тестирование, но зато негативное тестирование позволяет найти действительно непростые, коварные ошибки. Такие трудно обнаруживаемые ошибки часто проявляют себя в самый неожиданный момент и имеют значение на самом высоком уровне – когда что-то просто перестает работать. Кроме того, негативное тестирование позволяет протестировать функцию в критических условиях (параметрах) и выявить возможное ее неожиданное поведение.

Анализ покрытия кода

Покрытие кода – это метрика, указываемая обычно в процентах, призванная показать соотношение протестированного и непротестированного кода.

Значение покрытия кода говорит о том, сколько процентов кода было затронуто тестом. Чтобы покрыть больше кода, можно написать больше тестов. Покрытие кода подробно будет рассмотрено в главе 5, а на данный момент достаточно сказать, что покрытие кода — ключ к эффективному модульному тестированию.

Высокие значения покрытия кода могут вводить в заблуждение, и к ним надо относиться гораздо критичнее, чем к полученным низким значениям покрытия кода. В то же время модульное тестирование меньше чем с 50%-ым покрытием кода — это очень плохо; оптимальными значениями являются 60-80%. Стремиться к значениям больше 80% — не всегда нужно. Закон убывающей доходности[3] хорошо применим к покрытию кода с процентом от 80%. Не нужно пытаться получить 100%-ое покрытие, помните, что покрытие кода — это лишь одна из метрик, за которую ответственны разработчики, но не

3 Закон убывающей доходности — экономический закон, гласящий, что сверх определённых значений факторов производства (земля, труд, капитал) увеличение одного из этих факторов не обеспечивает эквивалентный прирост дохода, то есть доход растёт медленнее, чем фактор.

единственная. В главе 5 будет показано, как изменить и визуализировать покрытие кода.

Важно, чтобы информация о покрытии кода была сгенерирована автоматически без малейшего вмешательства человека. Процесс вычисления покрытия кода должен быть бесшовным.

4.6. Практическое тестирование

К сожалению, далеко не все функции так компактны, как функция `sum`. Но прошу заметить, что даже такая короткая функция с нулевой зависимостью оказалась не так проста в тестировании, как могло бы показаться на первый взгляд. Это снова демонстрирует сложность модульного тестирования в целом и модульного тестирования JavaScript в частности: если у простой надуманной функции есть столько «глюков», то что говорить о реальных функциях?

4.6.1. Работа с зависимостями

Ранее проблема зависимостей была уже затронута в этой книге. Почти у каждой функции есть внешние зависимости, которые нельзя протестировать модульным тестированием. Модульное тестирование может «купировать» зависимости, используя имитации и заглушки. Здесь речь идет именно о средствах модульного тестирования и временном «купировании» зависимостей, в то же время у автора тестируемого кода тоже есть средства извлечения зависимостей, так сказать, на постоянной основе, описанные в главе 2.

Прежде чем продолжить, давайте разберемся, в чем разница между имитациями (mock) и заглушками (stub). Имитации используются для команд, а заглушки используются для запросов (данных).

Дубли

Тестовые дубли — общее название для объектов (по аналогии с дублерами в кино), используемых тестами для имитации или заглушивания зависимостей. Дубли могут действовать как заглушки и как имитации одновременно, гарантируя, что внешние методы и API будут вызваны, задавая, сколько раз они были вызваны, перехватывая запрашиваемые параметры и возвращая требуемые отве-

ты. Тестовый дубль, который захватывает и записывает информацию о том, как вызывались методы, называется *шпионом*.

Mock-объекты (имитации)

Mock-объекты (объекты-имитации) используются для проверки того, что ваша функция корректно вызывает внешние API. Тесты с использованием имитаций (mock-объектов) позволяют проверить, что тестируемая функция передает корректные параметры (по типу или по значению) внешнему объекту. В случае консольных приложений (работающих в режиме командной строки), имитации тестируют команды. В мире событийно-ориентированных архитектур имитации позволяют тестировать генерируемые события.

Давайте рассмотрим такой пример:

```
// производственный код:
function buyNowClicked(event) {
        hub.fire('addToShoppingCart', { item: event.target.id });
}
Y.one('#products').delegate('click', buyNowClicked, '.buy-now-button');
/* ... в коде теста: */
testAddToShoppingCart: function() {
        var hub = Y.Mock();
        Y.Mock.expect(hub,
        {
          method: "fire"
          , args: [ "addToShoppingCart" , Y.Mock.Value.String]
        }
);
Y.one('.buy-now-button').simulate('click');
Y.Mock.verify(hub);
```

Данный код тестирует поток кнопки **Купить сейчас** (Buy Now!) (элемент, который содержит CSS-класс `buy-now-button`). При нажатии данной кнопки ожидается событие `addToShoppingCart`, которому должен быть передан строковой параметр. Для этого в нашем примере производится имитация концентратора событий, и метод `fire`, как и ожидалось, запускается с одним строковым параметром (ID кнопки **Купить сейчас**).

Заглушки

Заглушки (stub) используются для возврата фиксированных значений/объектов в тестируемую функцию в местах вызова внешних методов. Заглушку не интересует, как был вызван метод внешнего объекта, заглушка просто возвращает фиксированный объект, заданный вами заранее. Обычно в качестве такого объекта выступает какое-либо ожидаемое вызывающей стороной значение, если у нас с вами позитивное тестирование, или какое-то неожиданное/нетривиальное значение, если мы проводим негативное тестирование.

Тестирование с заглушками подразумевает, что вы заменяете ваши реальные объекты заглушками, которые будут возвращать фиксированные значения, заданные вами при написании тестового сценария.

Рассмотрим следующий код функции добавления в «Корзину»:

```
function addToShoppingCart(item) {
  if (item.inStock()) {
            this.cart.push(item);
           return item;
  } else {
           return null;
          }
}
```

Один из способов протестировать этот код — заглушить метод `item.inStok` и протестировать два возможных пути развития ситуации добавления в Корзину (успешный и неуспешный):

```
testAddOk: function() {
   var shoppingCart = new ShoppingCart()
   , item = { inStock: function() { return true; } }
   , back = shoppingCart.addToShoppingCart(item)
   ;
   Y.Assert.areSame(back, item, "Элемент не возвращен");
   Y.ArrayAssert.contains(item, "Элемент не помещен в корзину!");
}
, testAddNok: function() {
```

```
        var shoppingCart = new ShoppingCart()
        , item = { inStock: function() { return false; } }
        , back = shoppingCart.addToShoppingCart(item)
        ;
        Y.Assert.isNull(back, "Возвращенный элемент не Null ");
        Y.ArrayAssert.isEmpty(shoppingCart.cart, "Корзина не пуста!");
    }
```

Данное тестирование захватывает оба ответвления функции путем заглушки объекта `item`. Конечно, могут быть заглушены не только параметры, обычно глушатся или имитируются все внешние зависимости.

Шпионы

Тестовый шпион как бы окутывает «реальный» объект, переопределяя некоторые его методы, а со всеми остальными методами объекта позволяя работать как ни в чем не бывало (режим полной прозрачности, как будто шпиона и нет вовсе). При этом действия шпиона при перекрытии чаще всего сводятся либо к возврату некоторых фиксированных значений в местах вызова определенных методов объекта (с определенными параметрами), либо просто к подсчету того, сколько раз был вызван тот или иной метод. Остальные методы, которые не прерываются шпионом, работают в обычном режиме. Шпионы отлично подходят для объектов, которые содержат как «тяжелые» операции, которые вам нужно имитировать, так и легковесные операции, с которыми можно справиться без имитации.

В качестве небольшого отступления дам несколько рекомендаций:

- Вы не обязательно должны использовать только заглушки или только имитации, тут ограничений никаких нет. Можно использовать и то и другое. Более того, вы можете в рамках одной дублирующей конструкции заложить как некоторые функции имитации, так заглушки. Нужно гибридное решение? Создавайте!

- В тех случаях, когда вам нужно отследить, сколько раз была вызвана какая-либо внешняя функция, лучшее решение состоит в использовании тестового шпиона.

Возвращаясь к нашим баранам, то есть шпионам, рассмотрим пример тестового сценария с использованием шпиона для нашей функции `sum`:

```
, testWithSpy: function() {
  var origSum = sum
    , sumSpy = function(a, b) {
          Y.Assert.areSame(a, 2, 'первый аргумент = 2!');
          Y.Assert.areSame(a, 9, 'второй аргумент = 9!');
          eturn origSum(a, b);
    }
  ;
  sum = sumSpy;
  Y.Assert.areSame(sum(2, 9), 11
          , '2 + 9 does not equal 11?');
  sum = origSum; // сбрасываем ее
}
```

Данный шпион является локальным для этого теста, он просто проверяет, что переданные аргументы являются ожидаемыми. Хорошим стилем в использовании шпионов является подход, при котором шпион не должен вызывать базовый "реальный" метод, если это займет много системных ресурсов (например, сетевой вызов), и вместо этого он может просто вернуть фиксированные значения (как ожидаемые, в рамках позитивного тестирования, так и запредельные в рамках негативного тестирования (например, смоделировав отказ сети)). В таких случаях шпион проявляет себя в качестве заглушки.

В качестве инструмента, позволяющего эффективно создавать и использовать всевозможные тестовые дубли (имитации, заглушки) и шпионы, хотелось бы отметить фреймворк **Jasmine**. И чуть дальше в этой главе мы подробнее поговорим о нем.

4.7. Асинхронное тестирование

JavaScript в значительной мере опирается на события. Поэтому в рамках рассмотрения тестирования мы должны учесть этот факт. Для таких случаев (тестирования событийно-ориентированных приложений/взаимодействий) предусмотрено так называемое асинхронное тестирование.

Асинхронное тестирование приостанавливает поток тестирования в ожидании события, после чего возобновляет поток тестирования в обработчике событий.

Далее мы рассмотрим реализацию асинхронного тестирования средствами фреймворка YUI Test.

Использование YUI Test

Использование асинхронного тестирования средствами YUI Test выглядит примерно так:

```
testAsync: function () {
  var test = this, myButton = Y.one('#button');
  myButton.on('click', function() {
        this.resume(function() {
              Y.Assert.isTrue(true, 'Вы убили мой линкор!');
        });
}
myButton.simulate('click');
this.wait(2000);
}
```

Этот код имитирует нажатие кнопки и ждет две секунды. Если тест потом не будет продолжен, тест будет считаться неудачным; в противном случае тест будет продолжен (нужно надеяться, что успешно) и будет протестировано утверждение, заданное с помощью Y.Assert[4]. На практике мы можем делать более интересные вещи, чем просто имитация нажатия на кнопку и проверка, был ли вызван обработчик ее нажатия.

Если ваш код производит много изменений в стилях (CSS-таблицах) в качестве реакций на какие-либо события, то асинхронное тестирование — это то, что вам нужно. В то же время, для «тяжелого» тестирования пользовательских интерфейсов, интеграционного тестирования лучше всего использовать специальные инструменты типа Selenium, так как возможностей просто асинхронного тестирования вам может не хватить. Кроме того, не все DOM-события могут быть имитированы, так что для их тестирования вам потребуются другие типы тестирования.

4 Подробнее с утверждениями вы можете ознакомиться по ссылке http://yuilibrary.com/yui/docs/test/#assertions.

Несмотря на все это, асинхронное тестирование просто необходимо использовать при тестировании событийно-ориентированного кода типа такого:

```
hub.on('log', function(severity, message) {
        console.log(severity + ': ' + message);
});
```

Тестирование такого кода требует имитации глобального объекта `console` (это команда, а не запрос):

```
console = Y.Mock();
testLogHandler: function () {
        var sev = 'DEBUG', message = 'TEST';
        Y.Mock.expect(console, { method: log
                , arguments: [ sev, message ] });
        hub.fire('log', sev, message);
        this.wait(function() {
                Y.Mock.verify(console);
        }, 1000);
}
```

Здесь мы используем реальный концентратор событий, поскольку сейчас мы не тестируем функциональность самого концентратора. Данный тест проверяет, что метод `log` имитированного объекта вызывается с ожидаемыми аргументами.

4.8. Запуск тестов: сторона клиента

Допустим, вы написали ряд тестовых сценариев и поместили их в один из тестовых наборов. Что дальше? А дальше – запуск тестов в вашем браузере путем двойного щелчка мыши по связующему HTML-документу, содержащему связывающий код (связывающий тестовый сценарий и тестируемый код).

Это, конечно, прикольно, но автоматизацией здесь и не пахнет. А что это за сапожник без сапог: пишем код для автоматизации всего и вся, а свои собственные процессы автоматизировать не можем, что ли? Можем. Существует несколько стратегий по автоматизации процесса тестирования в рамках общего процесса разработки.

4.8.1. PhantomJS

PhantomJS (http://phantomjs.org/) – это WebKit-браузер, доступный и управляемый программно. По сути PhantomJS – это возможность работать с WebKit из консоли, используя JavaScript и без браузера. Это отличная «песочница», внутри которой вы можете производить модульное тестирование без необходимости запуска кода в реальном браузере или где-либо еще. Однако при этом PhantomJS даже не пытается обработать всевозможные CSS-слои, протестировать, как они реализованы в каждом браузере (хотя это было бы очень актуально).

Тестирование необычного или специфичного для браузера CSS требует наличия браузера, для которого писался CSS, а тут мы, наоборот, уходим от браузеров. Но, с другой стороны, модульное тестирование обычно фокусируется на базовой функциональности безотносительно интерфейса пользователя (за внешний вид которого отвечают CSS-таблицы стилей). И в этом-то нам и поможет PhantomJS, так как позволяет обработать и протестировать тонну всевозможных современных особенностей, «фич», за исключением разве что только каких-то совсем ультрасовременных CSS 3D, локальных хранилищ и WebGL. В то же время PhantomJS/WebKit постоянно развивается, поэтому если время от времени его обновлять, то, скорее всего, и поддержка вышеприведенных наворотов будет в него добавлена.

Современный PhantomJS больше не требует X-сервера для запуска, поэтому его установка стала гораздо проще. Просто загрузите исполнимые файлы для вашей операционной системы, и вы готовы к работе. Вы также можете скомпилировать `phantomjs` из исходного кода, но это займет много времени, поскольку также должна быть скомпилирована большая часть библиотеки Qt, а это монстр какой еще поискать.

Вот наш «сложный» JS-файл, предназначенный для тестирования `sum.js`:

```
YUI().add('sum', function(Y) {
        Y.MySum = function(a, b) { return a + b };
});
```

Как только вы все установите, выполнение ваших YUI-тестов через PhantomJS превратится в довольно простое занятие, но есть один нюанс. Наше связующее звено в виде HTML-файла требует небольших изменений:

```
<html>
 <head>
        <title>Тест суммы</title>
 </head>
 <body class="yui3-skin-sam">
        <div id="log"></div>
        <script src="http://yui.yahooapis.com/3.18.1/build/yui/yui-min.js">
        </script>
        <script src="sum.js"></script>
         <script src="phantomOutput.js"></script>
        <script src="sumTests.js"></script>
 </body>
</html>
```

Обратите внимание, что в этом случае загружается другой JavaScript-файл — `phantomOutput.js`, который определяет небольшой YUI-модуль `phantomjs`:

```
YUI().add('phantomjs', function(Y) {
 var TR;
 if (typeof(console) !== 'undefined') {
        TR = Y.Test.Runner;
         TR.subscribe(TR.COMPLETE_EVENT, function(obj) {
        console.log(Y.Test.Format.JUnitXML(obj.results));
         });
 }
});
```

Единственная цель этого модуля заключается в выводе на консоль результатов теста в формате XML JUnit после завершения тестов (YUI поддерживает и другие форматы вывода, которые можно использовать вместо формата XML JUnit). Вы вольны использовать тот формат, который понимает ваш инструмент сборки. Мой, например, Hudson/Jenkins понимает формат XML JUnit, поэтому я и использую его. Эта зависимость должна быть задекларирована в вашем тестовом файле. Рассмотрим `sumTests.js`:

```
YUI({
```

```
    logInclude: { TestRunner: true },
}).use('test', 'sum', 'console', 'phantomjs', function(Y) {
    var suite = new Y.Test.Suite('sum');
    suite.add(new Y.Test.Case({
            name:'simple test',
             testIntAdd : function () {
                    Y.log('testIntAdd');
                    Y.Assert.areEqual(Y.MySum(2 ,2), 4);
            },
      testStringAdd : function () {
        Y.log('testStringAdd');
        Y.Assert.areEqual(Y.MySum('my', 'sum'), 'mysum');
      }
    }));
    Y.Test.Runner.add(suite);
    // Инициализируем консоль
    var yconsole = new Y.Console({
      newestOnTop: false
    });
    yconsole.render('#log');
    Y.Test.Runner.run();
});
```

PhantomJS захватит консольный вывод и сохранит его в файле для дальнейшей обработки. К сожалению, включение модуля `phantomjs` в наши тесты сейчас не идеально. В главе 8 мы сделаем этот процесс более динамичным.

Далее приведен сценарий PhantomJS для захвата вывода теста:

```
var page = new WebPage();
page.onConsoleMessage = function(msg) {
        console.log(msg);
        phantom.exit(0);
};
page.open(phantom.args[0], function (status) {
          // Проверяем, успешно ли загружена страница
        if (status !== "success") {
                console.log("Unable to load file");
                phantom.exit(1);
         }
});
```

Все довольно просто. Данный сценарий PhantomJS в качестве параметра принимает URL, указывающий на HTML-файл «связующего звена», загружает его и, в случае успеха, ожидает консольного вывода для захвата.

Наш сценарий просто выводит «захваченный» вывод, но вы можете перенаправить вывод этого сценария в файл, или же сценарий PhantomJS может сам записать вывод в файл при необходимости.

У PhantomJS нет доступа к JavaScript, работающему на самой загруженной странице, таким образом, мы используем консоль для передачи тестового вывода страницы, загружаемой в PhantomJS, где вывод может быть сохранен.

Запустить наш тест можно командой:

```
% phantomjs ~/phantomOutput.js sumTests.html
```

Первый параметр — это JavaScript-файл теста, а второй — «связующее звено» (HTML-файл). Вывод будет примерно таким (хотя и не так хорошо отформатированным):

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
    <testsuite name="simple test" tests="2" failures="0" time="0.005">
        <testcase name="testIsObject" time="0"></testcase>
        <testcase name="testMessage" time="0.001"></testcase>
    </testsuite>
</testsuites>
```

Это и есть вывод в формате JUnit XML для двух созданных нами тестов.

Для полноты картины вы можете добавить сюда еще снимки экрана, причем сделать это довольно просто. Ниже приведен полный код тестового сценария с добавленной поддержкой снимков экрана. По сути, отличие только в том, что мы «прорисовываем» вывод после любого вывода на консоль.

```
var page = new WebPage();
page.viewportSize = { width: 1024, height: 768 };
page.onConsoleMessage = function(msg) {
  console.log(msg);
```

```
    setTimeout(function() {
    page.render('output.png');
    phantom.exit();
  }, 500);
};
page.open(phantom.args[0], function (status) {
    // Проверяем, успешно ли загрузилась страница
    if (status !== "success") {
      console.log("Невозможно загрузить файл");
      phantom.exit(1);
    }
});
```

Первым делом я установил размер виртуального окна, затем получил результаты прорисовки страницы в PNG-файл. Данный сценарий генерирует снимки экрана после каждого теста. Помимо PNG-формата PhantomJS также может создавать снимки экрана в форматах PDF и JPG. На рис. 4.2 вы можете увидеть наш снимок экрана.

Рис. 4.2. Снимок экрана PhantomJS

Превосходно! По умолчанию фон создаваемого изображения прозрачный, но мы можем нормально просмотреть вывод YUI Test, поместив его в HTML-элемент `<div id="log">`.

Постойте! А что насчет некоторых сообщений, отправляемых YUI службе протоколирования? Мы тоже хотим их захватить! Для этого нам нужно снова посетить YUI-модуль `phantomjs`:

```
YUI().add('phantomjs', function(Y) {
  var yconsole = new Y.Console();
```

```
yconsole.on('entry',
  function(obj) {
    console.log(JSON.stringify(obj.message));
  }
);
if (typeof(console) !== 'undefined') {
  var TR = Y.Test.Runner;
  TR.subscribe(TR.COMPLETE_EVENT, function(obj) {
    console.log(JSON.stringify(
      { results: Y.Test.Format.JUnitXML(obj.results) }));
  });
}
}, '1.0', { requires: [ 'console' ] });
```

Самое большое дополнение — объект `Y.Console`, который мы создали исключительно для захвата сообщений протоколирования YUI Test. Прослушка события `entry` даст нам все сообщения объекта, которые мы строколизировали с использованием JSON (обратите внимание, что объект JSON включен в WebKit, а наш JavaScript-код выполняет браузер PhantomJS/WebKit).

Теперь у нас два типа сообщений, выведенных на консоль, передаются «обратно» нашему сценарию PhantomJS: сообщения протоколирования и окончательные результаты JUnit XML. Код PhantomJS на стороне сервера должен отслеживать оба типа сообщений.

Вот первое сообщение:

```
{
  "time":"2015-02-23T02:38:03.222Z",
  "message":"Testing began at Wed Feb 22 2015 18:38:03 GMT-0800 (PST).",
  "category":"info",
  "sourceAndDetail":"TestRunner",
  "source":"TestRunner",
  "localTime":"18:38:03",
  "elapsedTime":24,
  "totalTime":24
}
```

Информацию о том, что означает то или иное поле сообщения, можно найти на веб-сайте YUI: http://yuilibrary.com/yui/docs/console/#anatomy.

Давайте рассмотрим обновленную функцию `onConsoleMessage` из сценария PhantomJS (эта функция — единственное, что изменилось в нашем сценарии):

```
page.onConsoleMessage = function(msg) {
    var obj = JSON.parse(msg);
    if (obj.results) {
        window.setTimeout(function () {
            console.log(obj.results);
            page.render('output.png');
            phantom.exit();
            }, 200);
    } else {
            console.log(msg);
    }
};
```

Если не учитывать JSON-разбор консольного вывода, этот сценарий получает результаты тестирования и сразу же завершает свое выполнение. При этом вы должны гарантировать, что ничто другое (например, другие ваши тесты или тестируемый код) ничего не пишет в файл `console.log`!

При особом желании можно получить снимок экрана раньше, чем до/во время/после каждого теста, чтобы получить фактический ход тестирования (вы будете знать, что делает каждый тест).

В завершение нашего разговора о PhantomJS хотелось бы отметить, что PhantomJS — превосходное средство модульного тестирования во время автоматического процесса сборки. Так, во время сборки вы можете передать этому сценарию список файлов/URL, которые будут выполнены в WebKit-браузере. В то же время не забывайте, что вам нужно выполнить тестирование и в реальных браузерах перед тем, как использовать ваш код в производственной среде. Но об этом мы поговорим в следующем разделе, посвященном Selenium'у.

4.8.2. Selenium

Тестирование в реальных браузерах

Используя Selenium Remote Control (RC) или Selenium2 (WebDriver), вы можете произвести модульное тестирование в реальных бра-

узерах: или в браузере, работающем на вашей удаленной машине, или же в удаленном браузере. Используя Selenium JAR на машине, где установлен Firefox, Safari, Chrome или IE, можно произвести модульное тестирование в любом из этих браузеров и сохранить результат этого тестирования. Selenium2/WebDriver — лучший выбор начинающего тестера, что касается Selenium RC, то его не рекомендую собирать, если вы только начинаете работать с Selenium, обратитесь к нему, уже обладая некоторым опытом работы с Selenium'ом.

Давайте разберемся, как это работает. Установите Selenium на машине, на которой установлен браузер, в котором вы будете производить тестирование. Для этого загрузите последнюю версию JAR Selenium с официального сайта: http://docs.seleniumhq.org/download/. На момент написания этих строк последней была версия 3.141. Теперь запустите Selenium:

```
% java -jar ~/selenium-server-standalone-3.141.0.jar
```

В результате будет запущен сервер Selenium (порт по умолчанию — 4444). Этот порт должен быть доступен с машины, на которой вы будете запускать клиент Selenium, поэтому настройте соответствующим образом ваш брандмауэр. Изменить порт можно с помощью опции `-port`.

Теперь, когда сервер Selenium установлен и запущен, вам нужно попросить его открыть браузер и получить URL. Это означает, что вы нуждаетесь в веб-сервере, который будет обслуживать ваши тестируемые файлы. Это нормально. При этом не обязательно скармливать Selenium'у весь код всего вашего JavaScript-приложения, достаточно только того, что должно быть протестировано.

В то же время лучше всего хранить код тестов в том же самом корневом каталоге (`DocumentRoot`), что и производственный код. Если вы не желаете размещать код тестов в производственном каталоге документов, нужно создать отдельный каталог в корневом каталоге документов — для файлов с тестами. Этот каталог будет содержать зеркало вашей производственной структуры каталогов и файлы тестов.

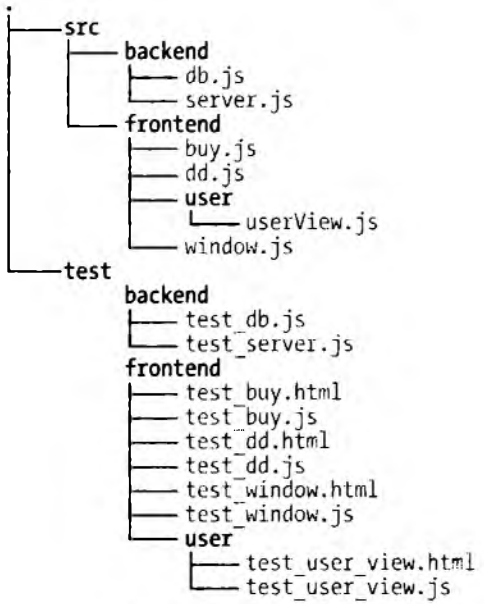


Рис. 4.3. Структура каталогов исходного кода

У вас должна получиться структура каталогов, подобная изображенной на рис. 4.3.

Как видно из рис. 4.3, каталог `test` зеркально отображает дерево `src`. Для каждого файла из каталога `src` в каталоге `test` есть соответствующий ему тестовый файл. Файл связующего звена, `test_usrt_view.html`, выглядит примерно так:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
    <head>
        <meta http-equiv="Content-Type"
                content="text/html; charset=ISO-8859-1">
        <title>User View Tests</title>
    </head>
    <body class="yui3-skin-sam">
        <h1>Test User View</h1>
        <div id="log" />
        <script src="http://yui.yahooapis.com/3.18.1/build/yui/yui-min.js">
        </script>
        <script src="../../../src/frontend/user/userView.js"></script>
```

```
        <script src="test_user_view.js"></script>
    </body>
</html>
```

В этом HTML используются относительные пути для указания тестируемого файла/модуля: `user_view.js`. При открытии этого HTML на веб-сервере будут найдены все необходимые файлы. Используя npm-пакет **webdriverjs**, мы можем легко отправить URL на сервер Selenium для выполнения:

```
var webdriverjs = require("webdriverjs")
, url = '...';
browser = webdriverjs.remote({
  host: 'localhost'
  , port: 4444
  , desiredCapabilities: { browserName: 'firefox' }
});
browser.init().url(url).end();
```

Данный код свяжется с сервером Selenium, запущенным на 4444-ом порту узла `localhost`, и попросит его запустить Firefox и загрузить указанный URL. Половина дела сделана! Теперь нам нужно захватить вывод теста — те самые полезные сообщения, выводимые YUI Test. Кроме того, мы хотим захватить снимок экрана и сохранить его локально.

Как и в случае с PhantomJS, мы должны так или иначе передать весь вывод (результаты теста, сообщения журнала, снимки экрана) обратно драйверу Selenium. В отличие от PhantomJS, Selenium не может удаленно считать консоль, поэтому нам нужно действовать иначе. Рассмотрим эквивалент модулю `phantomjs` для Selenium:

```
YUI().add('selenium', function(Y) {
 var messages = [];
    , yconsole = new Y.Console();
 yconsole.on('entry', function(obj) { messages.push(obj.message); });
 var TR = Y.Test.Runner;
 TR.subscribe(TR.COMPLETE_EVENT, function(obj) {
   // Данные для дампа
   var data = escape(JSON.stringify(
     {
       messages: messages
       , results: Y.Test.Format.JUnitXML(obj.results)
```

```
        }
    ));
    // Создаем новый узел
    var item = Y.Node.create('<div id="testresults"></div>');
    item.setContent(data);
    // Добавляем к документу
    Y.one('body').append(item);
  });
}, '1.0', { requires: [ 'console', 'node' ] });
```

Вы поняли, в чем фокус? Вместо того чтобы записывать все данные в `console.log`, мы сохраняем все сообщения, полученные от YUI Test, в массив. По завершении тестов мы создаем `<div>` с определенным ID и выводим JSON-версию всех сообщений и результатов. Наконец, мы добавляем этот новый элемент к текущему документу. Отметим два наиболее важных момента: во-первых, `<div>` должен быть видимым, а во-вторых, мы должны экранировать все содержимое результирующей JSON-строки, так как вывод в формате JUnit XML – это непривычно, непривычен XML. При этом мы не хотим, чтобы браузер анализировал такое содержимое, иначе оно будет потеряно как недопустимый HTML.

Теперь давайте посмотрим на другую половину, чтобы понять, как все это совместить воедино. Рассмотрим клиентскую часть Selenium:

```
var webdriverjs = require("webdriverjs")
  , url = '...'
  , browser = webdriverjs.remote({
      host: 'localhost'
      , port: 4444
      , desiredCapabilities: { browserName: 'firefox' }
  })
;
browser.init().url(url).waitFor('#testresults', 10000, function(found) {
  var res;
  if (found) {
    res = browser.getText('#testresults',
    function(text) {
      // Сделаем что-то большее, нежели console.log
      console.log(JSON.parse(unescape(text.value)));
    }
  );
```

```
    } else {
      console.log('РЕЗУЛЬТАТЫ ТЕСТА НЕ НАЙДЕНЫ');
    }
  }).end();
```

После загрузки URL мы ждем (`WaitFor`) элемента с ID `testresults` (мы будем ждать максимум 10 секунд). Когда он будет обнаружен, мы извлекаем его текст.

Первое, что нам нужно сделать, — это избавиться от экранирования нашего XML методом `unescape`. После этого мы можем воссоздать хэш, содержащий свойства `message`, которое является массивом сообщений YUI Test, и `results`, содержащее вывод JUnit XML для этого теста.

Таким образом вы можете запускать ваши тесты в реальных браузерах Firefox, Chrome или IE из командной строки (только нужно, чтобы на машине, на которой установлен и запущен сервер Selenium, были установлены эти браузеры).

Чтобы использовать Chrome, вам возможно потребуется добавить специальный драйвер для Chrome. Последнюю версию этого драйвера можно загрузить по адресу: https://code.google.com/p/chromium/downloads/list. Просто добавьте его в своей переменной окружения PATH и используйте имя `chrome` в качестве значения параметра `browserName` в хэше `desiredCapabilities`.

Теперь давайте поговорим о снимках экрана. У Selenium тоже есть поддержка снимков, поэтому вы тоже можете создавать картинки! В отличие от PhantomJS, более простой движок Selenium распознает только конец теста, а потому может создать снимок только после завершения теста.

Рассмотрим полный сценарий с поддержкой снимков. Обратите внимание: Selenium поддерживает только формат PNG. После получения результатов теста наш единственный снимок будет сохранен в локальный файл с именем `snapshot.png`:

```
var webdriverjs = require("webdriverjs")
  , url
  , browser = webdriverjs.remote({
      host: 'localhost'
```

```
        , port: 4444
        , desiredCapabilities: { browserName: 'firefox' }
    })
;
browser.init().url(url).waitFor('#testresults', 10000, function(found) {
    if (found) {
        var res = browser.getText('#testresults'
            ,function(text) {
              // Получаем сообщения журнала и результаты JUnit XML
              console.log(JSON.parse(unescape(text.value)));
             // Получаем снимок экрана
             browser.screenshot(
               function(screenshot) {
                 var fs = require('fs')
                    , filename = 'snapshot.png'
                    , imageData
                 ;
                 try {
                    imageData =
                       new Buffer(screenshot.value
                          , 'base64');
                    fs.writeFileSync(filename, imageData);
                 } catch(e) {
                    console.log('Ошибка получения снимка: '
                      + e);
                 }
               }
             );
            }
        );
    } else {
      console.log('РЕЗУЛЬТАТЫ ТЕСТА НЕ НАЙДЕНЫ!');
      }
}).end();
```

Во время сборки вы будете использовать более интеллектуальный способ передачи всех ваших тестов драйверам вместо указания имени файла или URL непосредственно в самом коде.

Основная проблема Selenium заключается в его медлительности: запуск сеанса браузера для каждого теста или набора тестов —

очень небыстрый процесс. Чтобы как-то помочь делу, можно воспользоваться проектом Ghost Driver (https://github.com/detro/ghostdriver/). Этот драйвер, подобно драйверу для Chrome, позволяет использовать PhantomJS в качестве браузера. Конечно, это существенно ускоряет запуск Selenium, но при этом поддерживаются не все примитивы Selenium. Однако если для вас производительность стоит на первом месте, то этот проект вам пригодится!

Другой способ повысить производительность Selenium — запускать тесты параллельно, используя сетку Selenium'ов. Мы рассмотрим этот способ запуска тестов в главе 5, где мы будем обсуждать покрытие кода, и в главах 6 и 8, где мы затронем другие типы тестирования и автоматизации соответственно.

Мобильное тестирование

С помощью Selenium вы можете производить модульное тестирование на своем телефоне или планшете. Давайте посмотрим, как именно.

iOS

Драйвер iOS Selenium работает или в симуляторе iPhone, или на физическом устройстве iPhone. Подробности доступны на сайте проекта: https://code.google.com/p/selenium/wiki/IPhoneDriver. Драйвер использует приложение `UIWebView`, чтобы принимать команды Selenium. Чтобы запускать тесты Selenium на физическом устройстве, вам нужен профиль Apple-разработчика, то есть вы должны быть членом программы разработчиков iOS (iOS Development Program), индивидуальное членство в которой стоило 99$ на момент написания этих строк.

Установите Xcode как минимум версии 4.2 (на момент написания этих строк уже была доступна версия 7), и вы сможете получить драйвер iPhone из данного Subversion-репозитория: http://selenium.googlecode.com/svn/trunk/.

Следуйте инструкциям установки (прилагаются к драйверу), и у вас в итоге должно собраться приложение WebDriver, которое можно будет потом установить в симулятор или непосредственно на «iУстройство». Вы можете подключиться к нему как к любому дру-

гому удаленному драйверу Selenium, поэтому симулятор, выполняющий приложение WebDriver, не должен быть на той же машине, на которой вы производите тестирование. Далее показано, как использовать `webdriverjs`:

```
var browser = webdriverjs.remote({
        host: 'localhost'
        , port: 3001
        , desiredCapabilities: { browserName: 'iPhone' } // или 'iPad'
        });
```

И теперь ваши тесты Selenium смогут работать в iOS. Обратите внимание на то, что мы используем порт 3001 — это порт по умолчанию для iPhone WebDriver. Аналогично, вы можете использовать iPad вместо iPhone.

Android

Подробности запуска Selenium WebDriver на Android доступны на сайте проекта (https://code.google.com/p/selenium/wiki/AndroidDriver). После загрузки Android SDK и установки эмулятора у вас есть два пути: запустить "стоковый" драйвер Selenium Android или же запустить какой-либо Android-ориентированный фреймворк тестирования. При этом, если вы запускаете одни и те же тесты на разных операционных системах, лучше использовать стандартный драйвер Selenium Android Web-Driver. Но, если все ваши тесты проводятся только в Android, вам лучше остановиться на решении с Android-ориентированным фреймворком тестирования (платформой тестирования).

Давайте поподробнее рассмотрим этот драйвер. Первым делом вам нужно откомпилировать .apk-файл и установить его на ваше Android-устройство или в симулятор Android. Подключение достаточно простое:

```
var browser = webdriverjs.remote({
        host: 'localhost'
        , port: 8080
        , desiredCapabilities: { browserName: 'android' }
});
```

Данный код предполагает, что эмулятор работает на локальной машине. Если это не так или же вы соединяетесь с реальным устрой-

ством, укажите IP-адрес своего Android-устройства. Если у вас есть проблемы с подключением или вообще появились какие-то вопросы, сходите по ссылке https://code.google.com/p/selenium/wiki/AndroidDriver, скорее всего, вам там помогут, и вы найдете необходимую информацию.

Еще было бы круто запустить Android-симулятор так, чтобы вы могли провести все свои тесты удаленно — с любой точки Земного шара. А вот вам решение: если вы запускаете эмулятор с опцией `-nowindow`, вы можете запустить приложение WebDriver, используя такой код (введите всю следующую команду в одной строке):

```
$ adb shell am start -a android.intent.action.MAIN -n
org.openqa.selenium.android.app.MainActivity
```

И вот теперь вы можете подключиться к нему локально или удаленно и произвести тестирование.

4.9. Тестирование: сторона сервера

Процесс модульного тестирования серверной части JavaScript-кода не очень отличается от процесса модульного тестирования какого-либо другого серверного кода. Тестовые сценарии и тестируемый код находятся и работают на одном и том же узле, что делает тестирование даже более простым, чем тестирование клиентской части. В этой главе для демонстрации тестирования серверной части кода мы будем использовать платформу модульного тестирования Jasmine.

Существует много других тестовых платформ, и вы можете использовать любую из них, если вам не нравится Jasmine. Кроме того, вы можете разработать и собственный тестировочный фреймворк, равно как и создать свой собственный язык программирования. Но задачи сейчас у нас другие, а потому мы не будем изобретать колесо и воспользуемся хорошим, на мой взгляд, решением – фреймворком Jasmine.

4.9.1. Фреймворк Jasmine

Установить Jasmine очень просто:

```
% npm install jasmine-node -g
```

Фреймворк Jasmine предоставляет нам возможность запуска тестов из командной строки, а также возможность получать результаты тестирования в разных форматах, а именно в человеко-читаемом и в предназначенном для автоматизированной обработки.

Основная идея состоит в том, что вы пишете набор тестов, используя предоставленный Jasmine синтаксис и семантику, а затем указываете Jasmine корневой каталог (или любой подкаталог), где находятся эти тесты. Jasmine рекурсивно «пройдется» по дереву каталогов и запустит все найденные в нем тесты.

По умолчанию Jasmine ожидает, что имена ваших тестовых файлов будут содержать строку `spec`, например `sum-spec.js`. Данное соглашение — наследство BDD[5], и переопределить его можно с помощью опции `--matchall`.

В рамках данной книги мы рассмотрим небольшие короткие примеры, основанные на уже полюбившемся примере – нашей реализации функции суммирования `sum`. Если вам нужно больше информации, перейдите на сайт Jasmine, чтобы ознакомиться со всеми ее возможностями: http://pivotal.github.io/jasmine/.

Итак, содержимое файла `mySum.js` у нас таково:

```
exports.sum = function(a, b) { return a + b };
```

Ниже приведен код Jasmine для тестирования целочисленного сложения:

```
var mySum = require('./mySum');
describe("Sum suite", function() {
  it("Adds Integers!", function() {
  expect(mySum.sum(7, 11)).toEqual(18);
  }
);
});
```

Запустить тест можно командой:

```
% jasmine-node .

Finished in 0.009 seconds
1 test, 1 assertion, 0 failures
```

5 Jasmine – это по сути BDD-фреймворк (Behavior-Driven Development — Разработка на Основе Поведения) для тестирования JavaScript-кода, позаимствовавший многие черты из RSpec.

Тест пройден! Матчеры Jasmine (в предыдущем примере — `toEqual`) эквивалентны утверждениям YUI Test. И вы можете сами писать свои матчеры.

Ключевое слово `expect` используется для задания ожиданий, которые будут проверяться по ходу тестирования. Конструкция `describe` предназначена для определения набора тестов, наборы могут быть вложенными (для определения теста внутри любого набора тестов используется конструкция `it`). Ключевые слова `describe` и `it` являются обычными вызовами функций, которым передаются два параметра. Первый — название группы или теста, второй — функция, содержащая код. Простой пример для ясности:

```
describe "Набор тестов", ->
        it "проверка ожиданий", ->
                expect(1 + 2).toBe(3)
```

Jasmine также поддерживает функции `beforeEach` и `afterEach`, которые выполняются до и после каждого тестового случая, аналогично функциям `setUp` и `tearDown` в YUI Test.

Зависимости

Факторинг зависимых объектов тестируемого Node.js[6]-кода может стать более требовательным, если зависимости помещаются посредством функции `require`. Вам нужно или изменить ваш исходный код для тестирования (что не очень хорошо), или имитировать (mock) функцию `require` (что очень сложно). К счастью, у нас есть отличный npm-пакет Mockery (http://bit.ly/XUdMoI). Пакет Mockery перехватывает все вызовы `require`, позволяя вставку имитированных версий зависимостей вашего кода вместо реальных.

Далее приведено описание нашей модифицированной функции `sum`, которая теперь читает JSON-строку из файла и добавляет свойства `a` и `b`, найденные в ней:

```
var fs = require('fs');
exports.sum = function(file) {
    var data = JSON.parse(fs.readFileSync(file, 'utf8'));
    return data.a + data.b;
};
```

6 Node.js – это асинхронная JavaScript бибилиотека для построения серверных приложений, которые используют конвенцию CommonJS.

Ниже приведен простой Jasmine-тест для нашей функции:

```
var mySum = require('./mySumFS');
  describe("Sum suite File", function() {
    it("Adds Integers!", function() {
    expect(mySum.sum("numbers")).toEqual(12);
  });
});
```

Сама JSON-строка находится в файле `numbers`:

```
{"a":5,"b":7}
```

Jasmine запустит тест, он будет пройден, и все будет хорошо. Но мы слишком много одновременно тестируем, а именно мы вовлекаем и используем зависимость `fs`, но суть модульного тестирования — в изолировании тестируемого кода. Мы не хотим, чтобы данная зависимость `fs` влияла на наши тесты. Конечно, это крайний случай, но давайте проследуем за логикой.

Используя Mockery как часть нашего Jasmine-теста, мы можем обработать вызов `require` и заменить модуль `fs` его имитированной версией. Рассмотрим наш новый Jasmine-сценарий:

```
var mockery = require('mockery');
mockery.enable();
describe("Sum suite File", function() {
  beforeEach(function() {
    mockery.registerAllowable('./mySumFS', true);
  });
  afterEach(function() {
    mockery.deregisterAllowable('./mySumFS');
  });
it("Adds Integers!", function() {
  var filename = "numbers"
    , fsMock = {
     readFileSync: function (path, encoding) {
       expect(path).toEqual(filename);
       expect(encoding).toEqual('utf8');
       return JSON.stringify({ a: 9, b: 3 });
     }
    }
  ;
```

```
    mockery.registerMock('fs', fsMock);
    var mySum = require('./mySumFS');
    expect(mySum.sum(filename)).toEqual(12);
    mockery.deregisterMock('fs');
});
```

Мы добавили и включили объект `mockery`, после чего мы указали Mockery игнорировать вызовы `require`. По умолчанию Mockery выведет ошибку, если какой-то требуемый модуль не будет известен ему (через `registerAllowable` или `registerMock`); но так как мы не хотим имитировать объект по ходу теста, то мы говорим Mockery замолчать.

Поскольку мы создаем новые имитации для модуля `fs` для каждого теста (скоро будет показано, зачем мы это делаем), нам нужно передать `true` в качестве второго параметра методу `registerAllowable`, чтобы метод `deregisterAllowable` работал надлежащим образом.

Наконец, давайте рассмотрим сам тест. В рамках него мы тестируем, что метод `fs.readFileSync` был вызван правильно и возвращает некоторые фиксированные данные обратно в функцию `sum`.

Вот тест для сложения строк:

```
it("Adds Strings!", function() {
    var filename = "strings"
        , fsMock = {
            readFileSync: function (path, encoding) {
                expect(path).toEqual(filename);
                expect(encoding).toEqual('utf8');
                return JSON.stringify({ a: 'testable'
                                      , b: 'JavaScript' });
            }
        }
    ;
    mockery.registerMock('fs', fsMock);
    var mySum = require('./mySumFS');
    expect(mySum.sum(filename)).toEqual('testableJavaScript');
    mockery.deregisterMock('fs');
});
```

Мы опять регистрируем новую имитацию для объекта `fs`, возвращающую фиксированный объект для конкатенации строк после проверки параметров `readFileSync`.

В качестве альтернативы, если бы мы хотели фактически прочитать JSON-объекты с диска, перед включением Mockery можно было бы создать реальный объект `fs` и мы бы делегировали вызовы из имитированного метода `readFileSync` к данному объекту — все это (процесс создания реального объекта и делегирование вызовов из имитированного метода) называется тестовым шпионом. Хранение данных тестирования в самих тестах очень полезно, особенно для таких небольших тестов, как наши.

Если имитируемые объекты будут большими или вам нужно будет хранить библиотеку таких объектов, вы можете указать Mockery заменить ваш оригинальный объект имитируемым с помощью следующего кода:

```
mockery.registerSubstitute('fs', 'fs-mock');
```

Как показано в документации по Mockery, теперь каждый вызов `require('fs')` на самом деле будет вызовом `require('fs-mock')`.

Mockery — отличный инструмент. Просто помните, что после его активации все вызовы `require` будут перенаправляться через Mockery!

Шпионы

Шпионы в Jasmine в основном используются для внедрений (инъекций) кода. При этом они обычно и заглушки, и имитации, все в одном лице.

Итак, допустим мы решили, что наша функция `sum`, читающая JSON-строку из файла, и функция `sum`, читающая операнды из списка параметра, могут быть объединены так:

```
exports.sum = function(func, data) {
        var data = JSON.parse(func.apply(this, data));
        return data.a + data.b;
};
exports.getByFile = function(file) {
```

```
        var fs = require('fs');
        return fs.readFileSync(file, 'utf8');
    };
    exports.getByParam = function(a, b) {
        return JSON.stringify({a: a, b: b});
    };
```

Мы обобщили ввод данных, и теперь операция суммирования у нас по факту присутствует только в одном месте. Во-первых, мы разделили операции получения данных и операции обработки данных. Во-вторых, благодаря этому мы можем удобно добавлять новые операции (вычитание, умножение и т.д.) над введенными данными.

Наш spec-файл для этого кода (без Mockery-содержимого для ясности) будет выглядеть примерно так:

```
mySum = require('./mySumFunc');
    describe("Sum suite Functions", function() {
        it("Adds By Param!", function() {
            var sum = mySum.sum(mySum.getByParam, [6,6]));
            expect(sum).toEqual(12);
    });
    it("Adds By File!", function() {
        var sum = mySum.sum(mySum.getByFile, ["strings"]));
        expect(sum).toEqual("testableJavaScript");
    });
});
```

И конечно же, файл со строками будет содержать следующее:

```
{"a":"testable","b":"JavaScript"}
```

Что-то здесь не совсем правильно. У нас нет хорошей изоляции функции `mySum`. Конечно, нам нужны отдельные тесты для разных способов ввода функции `mySum`: `getByParam` и `getByFile`. Но для тестирования самой функции `Sum` нам нужно имитировать и заглушить эти функции, чтобы функция суммы тестировалась в изоляции.

Такая ситуация — отличное место для применения шпионов Jasmine. Используя шпионы, мы можем протестировать две вспомогательные функции, чтобы убедиться, что они будут вызываться с корректными параметрами, а также позволить тестовому коду или вер-

нуть вызываемые значения, или передать вызов низлежащей функции.

Рассмотрим модифицированную версию функции со шпионом для `getByParam`:

```
it("Adds By Param!", function() {
        var params = [ 8, 4 ]
                , expected = 12
        ;
        spyOn(mySum, 'getByParam').andCallThrough();
        expect(mySum.sum(mySum.getByParam, params)).toEqual(expected);
        expect(mySum.getByParam).toHaveBeenCalled();
        expect(mySum.getByParam.mostRecentCall.args).toEqual(params);
});
```

После установки шпиона на `mySum.getByParam` и перенаправления его на вызов к актуальной реализации, мы можем проверить, что вспомогательная функция вызвана с корректными параметрами.

Замена строки `andCallThrough` следующей строкой:

```
spyOn(mySum, 'getByParam').andReturn('{"a":8,"b":4}');
```

позволит нашему тестовому сценарию вернуть фиксированные данные и пропустить вызов `getByParam`. Вы также можете предоставить альтернативную функцию, возвращаемое значение которой будет передано обратно вызывающей стороне. Это очень удобно, если нужно вычислить возвращаемое шпионом значение.

У Jasmine есть большое количество трюков, в том числе простое отключение теста, имитация `setTimer` и `setInterval` для более простого тестирования, поддержка асинхронного тестирования. Со всем этим вы можете познакомиться, перейдя на домашнюю страницу Jasmine. Кроме того, Jasmine может использоваться не только для тестирования серверной части JavaScript, этот фреймворк можно применять для модульного тестирования и клиентского JavaScript. В то же время извлечение вывода теста при выполнении Jasmine в браузере не так просто, как в случае YUI Test.

Вывод результатов тестирования

По умолчанию Jasmine выводит результаты теста на экран, что хорошо при разработке тестов, но не очень хорошо для автоматизированных запусков (в рамках автоматизированного тестирования, когда результаты предназначены для анализа какой-либо программой, а не человеком). Используя параметр `-junitreport`, вы можете заставить Jasmine выводить результаты тестов в широко используемом формате JUnit XML. По умолчанию все результаты тестов записываются в каталог `reports`, название файла соответствует названию тестового набора (первый параметр описания метода). Например:

```
% ls reports
TEST-Sumsuite.xml TEST-SumsuiteFile.xml
% cat reports/TEST-SumsuiteFile.xml
<?xml version="1.0" encoding="UTF-8" ?>
<testsuites>
<testsuite name="Sum suite File" errors="0" tests="2" failures="0"
time="0.001" timestamp="2014-07-25T15:31:48">
  <testcase classname="Sum suite File" name="Adds By Param!" time="0.001">
  </testcase>
  <testcase classname="Sum suite File" name="Adds By File!" time="0">
  </testcase>
</testsuite>
</testsuites>
```

Теперь наш вывод готов для импорта во что-то, что понимает этот формат, как будет показано в главе 8. Обратите внимание, число тестов, о которых будет отчитываться Jasmine, — это число функций `it`, а не число вызовов `expect`.

Резюме

Модульное тестирование JavaScript-кода в принципе несложное. В вашем распоряжении отличные инструменты, которые помогут вам как при написании, так и при выполнении unit-тестов. В этой главе было рассмотрено два таких инструмента — YUI Tests и Jasmine. Оба инструмента предоставляют полнофункциональные окружения для модульного тестирования JavaScript и предназначены для использования разработчиками, регулярно добавляющими новые опции и исправляющими ошибки.

Используйте блоки комментариев в начале функций для описания того, что это за функция, какие параметры использует и что она должна возвращать в качестве результата.

Помните, что слабое связывание делает имитацию и заглушку объектов значительно проще.

После выбора полнофункциональной платформы (фреймворка) модульного тестирования написание и выполнение тестов должны быть относительно безболезненными. Различные платформы предоставляют различные функции, поэтому убедитесь, что выбрали платформу, поддерживающую автоматизированный процесс сборки и поддерживающую различные необходимые вам вспомогательные инструменты тестирования (вероятно, вам понадобятся: асинхронное тестирование, имитация и заглушка зависимостей).

Для локального или удаленного запуска ваших тестов проще использовать PhantomJS или Selenium. PhantomJS предоставляет полноценный WebKit-браузер, а Selenium предоставляет базовый браузер плюс доступ к любому "реальному" браузеру, в том числе к браузерам iOS и Android.

Создание отчетов покрытия кода позволяет изменить область покрытия и эффективность ваших тестов. Об этом мы и поговорим в следующих главах.

Глава 5.

Покрытие кода

Несмотря на то, что метрика покрытия кода при тестировании является достаточно неоднозначной и требует вдумчивого анализа, она очень полезна и от нее не нужно отказываться. Изначально покрытие кода ассоциируется только с модульным тестированием, однако можно вычислить метрику покрытия кода и для интеграционного тестирования (обычно интеграционное тестирование проводится после модульного и предшествует системному тестированию). Также можно скомбинировать множество отчетов покрытия кода в один общий отчет, содержащий данные о модульном и интеграционном тестировании, что обеспечит полную картину покрытия кода вашим набором тестов.

Независимо от того, какие инструменты вы будете использовать для определения покрытия, последовательность действий всегда одна и та же: нужно подготовить JavaScript-файлы, для которых требуется определить покрытие кода, развернуть эти файлы, получить результаты покрытия и сохранить их в локальном файле, при необходимости скомбинировав результаты по разным тестам. Результаты могут выдаваться либо в виде чисел (процентов), либо в виде некоторого мило оформленного HTML-файла, но это уже зависит от ваших предпочтений и вашего инструмента.

5.1. Основы покрытия кода

Покрытие кода показывает, выполняется или нет та или иная строка кода в ходе тестирования, а если выполняется, то сколько раз. Это полезно для измерения эффективности ваших тестов. Теоретически, чем больше строк кода «покрыто», тем более полными являются ваши тесты.

Ниже приведена простая функция Node.js, возвращающая текущую цену акции для заданного символа (тикера):

```
/**
* Возвращает текущую цену акции для заданного символа (тикера)
* при обратном вызове
*
* @method getPrice
* @param symbol <String> тикер
* @param cb <Function> обратный вызов с результатами cb(error, value)
* @param httpObj <HTTP> Необязательный HTTP-объект для введения
* @return ничего
**/
function getPrice(symbol, cb, httpObj) {
   var http = httpObj || require('http')
      , options = {
        host: 'download.finance.yahoo.com' // Спасибо Yahoo!
        , path: '/d/quotes.csv?s=' + symbol + '&f=l1'
      }
   ;
   http.get(options, function(res) {
      res.on('data', function(d) {
      cb(null, d);
      });
   }).on('error', function(e) {
      cb(e.message);
     });
}
```

Дан тикер и обратный вызов. Данная функция получит текущую цену тикера. Она следует стандартному соглашению для обратных вызовов, согласно которому параметр ошибки задается первым. Также обратите внимание, что эта функция позволяет внедрять (заглушать) объект `http` для более простого тестирования, что уменьшает связывание между этой функцией и объектом `http`, улучшая тестируемость (вы также можете использовать Mockery для этого, см. гл. 4).

Внедрение зависимостей мы уже рассмотрели в главе 2, и здесь мы рассмотрим еще один пример, где введение зависимости сделает проще сопровождение кода. Что если хост, предоставляющий сервис курса акций, требует протокол HTTPS? Или HTTPS требует-

ся для одних акций, а HTTP – для других? Гибкость, предоставляемая инъекциями (injection) в таких случаях, будет очень к месту и отлично подойдет в данной ситуации.

Написание теста для максимального покрытия кода — почти всегда тривиальный процесс. Первым делом нам нужно заглушить объект `http`:

```
/**
* Простая заглушка для объекта HTTP
*/
var events = require('events').EventEmitter
  , util = require('util')
  , myhttp = function() { // Макет объекта 'http'
    var _this = this ;
    events.call(this);
    this.get = function(options, cb) {
      cb(_this);
      return _this;
    };
  }
  ;
util.inherits(myhttp, events);
```

Важно понять, что мы не тестируем модуль `http`, предоставленный Node.js. Да, в нем могут быть ошибки, но мы не пытаемся найти их. Нам нужно протестировать только код нашей функции, а не какой-либо внешний объект, независимо от того, кто его написал. Интеграционные тесты найдут любые ошибки взаимодействия между нашим кодом и его зависимостями.

После того, как мы заглушим объект `http`, можно приступать к тестированию. Далее показано, как мы можем использовать асинхронные методы тестирования YUI Tests (я опускаю шаблонное содержимое тестового кода для ясности):

```
testPrice: function() {
    var symbol = 'YHOO'
        , stockPrice = 50     // Принятие желаемого за действительное??
        , _this = this
        , http = new myhttp()
    ;
```

```
        getPrice(symbol, function(err, price) {
                _this.resume(function() {
                        Y.Assert.areEqual(stockPrice, price, "Цены не равны!");
                }, http); // Вводим наш объект 'http'
        http.fire('data', stockPrice); // Наши "поддельные" данные
        this.wait(1000);
}
```

Это базовый тест для успешного случая. Для тестирования случая ошибки нужно отправить событие `error`:

```
testPriceError: function() {
        var symbol = 'YHOO'
                , _this = this
                , http = new myhttp()
        ;
        getPrice(symbol, function(err, price) {
                _this.resume(function() {
                        Y.Assert.areEqual(err, 'ошибка', "Нет ошибки!");
                }, http);
        http.fire('error', { message: 'ошибка'} );
        this.wait(1000);
}
```

С этими двумя тестами метрика покрытия кода покажет 100%-ное покрытие кода — это удивительно! И что же это значит? Значит ли это, что больше не нужно тестировать функцию `getPrice`, поскольку она работает правильно, как и планировалось? Ну, конечно же, нет.

Даже если большинство (или даже все) строк выполнены тестом или набором тестов, это не обязательно означает, что были протестированы все граничные случаи или что даже общий случай был полностью протестирован во всем своем многообразии. Именно поэтому я и сказал в самом начале главы, что высокие проценты покрытия кода могут вводить в заблуждение. В данном случае мы запустили все строки кода, но не протестировали все возможные варианты запуска кода. Наша функция не полностью протестирована, но процент покрытия кода говорит об обратном. Таким образом, получается, что даже если мы достигли 100%-го покрытия кода, мы не можем сказать с уверенностью, что функция `getPrice` на 100% надежна.

В то же время низкие значения покрытия кода вам очень помогут в работе, так как дают ясно понять, что просто-напросто далеко не все строки кода были протестированы и вам еще рано разбираться с вариативностью исполнения одного и того же кода: у вас некоторые фрагменты остались вообще не охвачены тестированием.

Еще интересно сравнить объем кода тестового сценария и объем тестируемого кода. Размер написанного нами кода теста почти в три раза превышает объем тестируемой функции, а на самом деле даже больше, если посчитать еще и содержимое тестового шаблона YUI Test, которое не включено в эти листинги.

5.2. Данные покрытия кода

Значение метрики о покрытии кода обычно формируется на основании двух показателей: количества «покрытых» строк и количества «покрытых» функций, оба из которых выражаются в процентах.

Все это понятно для отдельного модульного теста, предназначенного для тестирования отдельно взятого модуля. Тонкость же в том, что если вы захотите протестировать только одну функцию или только один метод в рамках какого-то объекта, при вычислении процента все равно будет учитываться (стоять в знаменателе) полное количество строк кода. И значение покрытия кода будет низким. Аналогично, если ваши тесты рассредоточены в нескольких файлах при тестировании какого-то одного модуля, то покрытие теста модуля будет низким для каждого отдельного файла. И чтобы получить истинное значение покрытия кода, вам нужно сложить значения покрытия для каждого отдельного теста.

Продолжая эту нить рассуждений, при попытке вычислить общий процент покрытия кода всего приложения (или просто каталога файлов) полученное значение по умолчанию будет вычислено без учета файлов, для которых не написаны тесты. Метрики покрытия совершенно не учитывают файлы, с которыми не связаны тесты, поэтому еще раз взываю к осторожности при анализе «итогового» процента покрытия.

Кстати, решение проблемы игнорирования файлов, для которых не написаны тесты, при определении покрытия кода состоит в том, что вы должны сгенерировать «болванки» или «пустые» тестовые файлы для каждого файла, не требующего тестирования. Подробно данная техника будет обсуждаться чуть ниже в данной главе.

5.3. Практика определения покрытия кода

В этом разделе я продемонстрирую определение покрытия кода с использованием утилит покрытия YUI (http://yuilibrary.com/download/yuitest/). Утилиты определения покрытия входят в основной архив фреймворка YUI, так что если он у вас установлен, то ничего дополнительно качать и устанавливать не надо. А если не установлен, то скачайте его по вышеприведенной ссылке. В папке YUI перейдите в каталог `java/build`, и вы увидите в нем необходимые JAR-файлы: `yuitest-coverage.jar` и `yuitest-coverage-report.jar`. Для их запуска вам, конечно же, потребуется одна из последних версий Java (1.5 или новее), путь к которой указан в переменной окружения `PATH`.

Первый JAR-файл (`yuitest-coverage.jar`) используется для преобразования обычных JavaScript-файлов в инструментированные файлы для определения покрытия кода, а второй (`yuitest-coverage-report.jar`) генерирует отчеты из полученной информации покрытия.

Инструментирование файлов

Инструментировать файл очень просто:

```
% java -jar yuitest-coverage.jar -o coveraged.js instrument-me.js
```

Теперь файл `coveraged.js` — полная рабочая копия файла `instrument-me.js` со встроенным отслеживанием метрики покрытия кода. Если вы откроете данный файл, вы увидите, как он был преобразован: между каждым оператором будет вставлен новый оператор, постепенно увеличивающий счетчик, принадлежащий глобальной переменной, отслеживающей, выполнялась ли та или иная строка кода. Этим инструментованным файлом при тестировании заменяется исходный файл. После выполнения кода значе-

ние о количестве выполненных строк кода можно извлечь из глобальной переменной `_yuitest_coverage`.

Процесс получения информации о покрытии кода состоит из следующих этапов: инструментирование кода, выполнение тестов и извлечение информации о покрытии.

Извлечение информации о покрытии кода от интеграционных тестов требует инструментирования всех JavaScript-файлов приложения, развертывания пакета файлов, запуска тестов Selenium (или выполнения их вручную, но надеюсь, что у нас все это автоматизировано) и извлечения информации о покрытии.

Инструментировать весь код можно с помощью одной команды (вся эта команда должна быть введена в одну строку):

```
% find build_dir -name "*.js" -exec echo "Покрытие {}" \;
-exec java -jar yuitest-coverage.jar -o /tmp/o {} \;
-exec mv /tmp/o {} \;
```

Разберемся, что делает данная команда. Мы начинаем поиск всех JavaScript-файлов с корневого каталога развертывания, затем инструментируем все найденные JavaScript-файлы (заменяем изначальные версии файлов на инструментированные). При желании мы можем упаковать и развернуть приложение как обычно. Однако поскольку покрытие кода работает со строками кода, а не с операторами, убедитесь, что вы этого не сделали и используете инструменты определения покрытия на несжатом JavaScript-коде.

Анатомия файла покрытия

Инструмент покрытия YUI принимает JavaScript-файл и превращает его в файл, отслеживающий, какие строки и функции в нем выполняются, тем самым инструментируя его. Чтобы увидеть это преобразование в действии, давайте еще раз обратимся к функции `sum`:

```
function sum(a, b) {
    return a + b;
```

```
}
```

Для инструментирования этого файла выполните следующую команду:

```
% java -jar yuitest_coverage sum.js -o sum_cover.js
```

В результате преобразования нашего файла будет получен файл с 37 строками! Первые 18 строк — это шаблон, определяющий глобальные переменные и функции, которые использует YUI Test Coverage для отслеживания информации покрытия.

Оставшиеся 19 строк будут такими:

```
_yuitest_coverage["sum.js"] = {
    lines: {},
    functions: {},
    coveredLines: 0,
    calledLines: 0,
    coveredFunctions: 0,
    calledFunctions: 0,
    path: "/home/trostler/sum.js",
    code: []
};
_yuitest_coverage["sum.js"].code=["function sum(a, b)
  {"," return a + b;","}"];
_yuitest_coverage["sum.js"].lines = {"1":0,"2":0};
_yuitest_coverage["sum.js"].functions = {"sum:1":0};
_yuitest_coverage["sum.js"].coveredLines = 2;
_yuitest_coverage["sum.js"].coveredFunctions = 1;
_yuitest_coverline("sum.js", 1); function sum(a, b) {
  _yuitest_coverfunc("sum.js", "sum", 1);
  _yuitest_coverline("sum.js", 2); return a + b;
}
```

Первый блок строк определяет объект `_yuitest_coverage["sum.js"]`, который будет хранить счетчики строк и функций. В последнем блоке строк определены счетчики строк (есть две счетные строки в файле, строка 1 и строка 2, и одна функция в строке 1, которая называется `sum`). Изменение счетчиков происходит при каждом вызове `yuitest_coverline` или `_yuitest_coverfunc`. Эти функции

определены в шаблоне, и все их предназначение состоит в том, чтобы увеличивать счетчик строк или функций соответственно.

Итак, в нашем примере, как только этот файл будет загружен, будет вызвана функция `yuitest_coverline("sum.js", 1)`, поскольку была выполнена первая строка. Когда будет вызвана функция `sum`, впервые будет запущена функция `_yuitest_coverfunc`, затем будет вызвана `_yuitest_coverline`, увеличивающая счетчик строк, и, наконец, будет вызвана инструкция `return a + b`.

Таким образом, в начале каждого файла, между каждым оператором и в начале каждой функции YUI Test Coverage отслеживает строки и вызываемые функции.

Важно отметить, что при каждой новой загрузке страницы счетчики сбрасываются. Кэширование может предотвратить это, но, тем не менее, если файл был загружен несколько раз, предыдущая информация о покрытии кода все равно будет потеряна. Всегда лучше извлекать результаты покрытия сразу после выполнения каждого теста.

5.4. Развертывание

Для создания файлов покрытия клиентского и серверного кода используются различные стратегии. HTTP-сервер может динамически обрабатывать файлы покрытия на основе строки запроса или другого метода. Далее мы рассмотрим стратегии создания файлов покрытия для клиентского и серверного кода.

Развертывание JavaScript-клиента

В рамках модульного тестирования клиентского JavaScript-кода в связующем HTML-файле мы должны указать версию файла со счетчиками покрытия вместо обычной версии. Сделать это можно несколькими способами, в том числе и автоматизированным методом, который будет рассмотрен нами в главе 6. Самый простой способ заключается в том, чтобы инструментировать весь ваш код перед выполнением модульного тестирования. При выполнении теста HTML «подхватит» инструментованную версию вашего кода, поскольку имена файлов остались неизменными.

Есть и альтернативное решение: если вы используете веб-сервер Apache, можно динамически инструментовать код с помощью правил `mod_rewrite`. В этом случае нужно пометить JavaScript-файлы, для которых нужно получить информацию о покрытии в HTML. Для этого удобнее всего использовать строку запроса:

```
<script src="/path/to/file/myModule.js?coverage=1"></script>
```

Соответствие правилу `mod_rewrite` перенаправит запросы этого типа сценарию, который примет исходный файл, сгенерирует для него файл покрытия и вернет его вместо обычной версии:

```
RewriteEngine On
RewriteCond %{QUERY_STRING} coverage=1
RewriteRule ^(.*)$ make_coverage.pl?file=%{DOCUMENT_ROOT}/$1 [L]
```

В этом случае любой запрос файла с подстрокой `coverage=1` в строке запроса будет перенаправлен сценарию `make_coverage.pl`, который вернет файл покрытия для запрошенного файла.

Сценарий `make_coverage.pl` предельно прост:

```
#!/usr/bin/perl
use CGI;
my $q = CGI->new;
my $file = $q->param('file');
system("java -jar /path/to/yuitest_coverage.jar -o /tmp/$$.js $file");
print $q->header('application/JavaScript');
open(C, "/tmp/$$.js");
print <C>;
```

Если у вашего модуля есть внешние зависимости, которые также должны быть включены для выполнения ваших тестов, вы можете также инструментовать и этот внешний код, чтобы видеть связанность вашего кода. Однако я не рекомендую делать это! Суть модульного тестирования – изолировать модуль и проверить, как он работает, а не в том, чтобы увидеть, что другие модули могут использовать ваш модуль.

В идеальном случае никакие внешние зависимости не должны загружаться при тестировании одного модуля. Их нужно или заглушить, или имитировать в коде теста. Мало того, есть хуже ситуации, когда вы вынуждены отлаживать другой модуль, выходящий за рамки тестируемого в данный момент.

Что касается развертывания файлов покрытия для интеграционного/Selenium-тестирования, в этом случае решение не может быть простым. В этом случае весь код должен быть инструментован и затем развернут, как обычно. Обратите внимание на то, что инструментованный код будет работать более медленно, потому что у него в два раза больше операторов, поэтому не нужно проводить тесты производительности на таком коде.

Как только вы развернете код, то запустите свои тесты, при этом напоминаю, что информация о покрытии не сохраняется при перезагрузках. Перед перезапуском браузера или переходом на другую страницу необходимо сначала извлечь и сохранить информацию о покрытии кода.

К счастью, Selenium упрощает эту задачу, поскольку у каждого тестового сценария или набора тестов есть функция `tearDown`, подходящая для нашей задачи.

Кроме того, развернутая инструментованная сборка будет вам очень полезна при ручном тестирования. Загрузите страницу в своем браузере и щелкните где-нибудь. После этого вы сможете вывести информацию о покрытии на консоль (просмотрите глобальную переменную `_yuitest_coverage`), затем скопируйте и вставьте ее в файл для преобразования в HTML. Таким образом вы точно сможете увидеть, какой код был выполнен во время вашего случайного щелчка.

Важно отметить, что при этом щелчке вы по сути ничего не тестируете в обычном понимании этого слова. Вы просто узнаете, какой код был выполнен.

Развертывание JavaScript сервера

Имитация (mock) загрузчика Node.js — не слишком плохой способ динамически вводить код со счетчиками покрытия в тестируемый JavaScript. Если такой способ оказывается слишком страшным для вас (он включает переопределение приватного метода), есть другой способ.

Сейчас мы будем говорить о методе `Module_load` (Node.js) и его переопределении. Это внутренний метод (что, вообще говоря, призывает к осторожности), который отличается большой прозрачностью и тем самым будет нам очень полезен в рамках нашей задачи.

Давайте рассмотрим базовый код:

```
var Module = require('module')
        , path = require('path')
        , originalLoader = Module._load
        , coverageBase = '/tmp'
        , COVERAGE_ME = []
;
Module._load = coverageLoader;
// COVERAGE_ME содержит список файлов, для которых нужно сгенерировать
// файлы покрытия
// Эти JS-файлы будут запущены через yuitest-coverage.jar
// Вывод будет сохранен в coverageBase
// Потом будут запущены тесты
// Все вызовы 'require' будут проходить через это:
function coverageLoader(request, parent, isMain) {
        if (COVERAGE_ME[request]) {
                request = PATH.join(coverageBase, path.basename(request));
        }
        return originalLoader(request, parent, isMain);
}
// По окончании просмотрите глобальную переменную _yuitest_coverage
```

Сначала мы определяем, для каких JavaScript-файлов нужно создать файлы покрытия, а затем мы генерируем файлы покрытия для этих файлов в каталоге `/tmp`.

После этого мы выполняем наши тесты, используя любой тестировочный фреймворк (платформу тестирования). При выполнении наших тестов все их вызовы `require` будут пропускаться через функцию `coverageLoader`. Если запрошен файл, который мы хотим «померять», будет возвращена его версия со счетчиками покрытия кода (так называемый файл покрытия), в противном случае мы обращаемся к обычному загрузчику Node.js, чтобы загрузить запрошенный модуль в обычном режиме.

Когда все тесты будут выполнены, станет доступна глобальная переменная `_yuitest_coverage` (она будет преобразована в формат LCOV).

В вышеприведенном коде клиентский JavaScript говорит HTTP-серверу, чтобы тот сгенерировал файлы покрытия на основании параметра строки запроса. Теперь нам надо добавить дополнительный параметр к строке запроса (вызова) `require`, который будет сигнализировать серверному коду, надо или не надо возвращать файл покрытия для вызываемого файла.

Обратите внимание, что все это требует двух проходов: первый проход определяет, для каких файлов нужно сгенерировать файлы покрытия, а второй проход выполняет тесты, динамически перехватывает вызовы `require` и возвращает файлы покрытия для запрошенных файлов. Это происходит потому, что код `yuitest_coverage` требует порождения внешнего асинхронного процесса для создания файлов покрытия, а вызов `require` — синхронный. В этом нет ничего страшного, но об этом вы должны знать. Если Node.js даже освободит синхронный метод или если полностью синхронный генератор покрытия станет доступным, файлы покрытия могут быть сгенерированы в переопределенном методе `load`.

Итак, как же добавить дополнительный параметр в `request` для запроса покрытого кода? Для начала, ваш тестовый код может выглядеть как-то так:

```
my moduleToTest = require('./src/testMe', true);
```

Данный оператор внутри тестового сценария в случае вызова внешнего модуля добавляет второй параметр (`true`) к вызову `require`. Node.js проигнорирует этот неожиданный параметр, а мы его перехватим с помощью следующего регулярного выражения:

```
/require\s*\(\s*['"]([^'"]+)['"]\s*,\s*true\s*\)/g
```

Все это выглядит сложнее, чем есть на самом деле. А на самом деле все очень просто: идея заключается в том, чтобы загрузить ваш JavaScript-файл, а потом с помощью этого регулярного выражения перехватить все вызовы модулей с дополнительным параметром `true`. Можно использовать и другие способы (JavaScript-парсер, построить абстрактное синтаксическое дерево с помощью JSLint или Uglify.js, может быть что-то еще), но на практике приведенное регулярное выражение является самым адекватным и на 100% оправдало свое применение.

Как только вы получите список модулей, для которых нужно определить значение метрики покрытия кода, следующий код сгенерирует файлы покрытия для этих модулей:

```
var tempFile = PATH.join(coverageBase, PATH.basename(file));
        , realFile = require.resolve(file)
;
exec('java -jar ' + coverageJar + " -o " + tempFile + " " + realFile
        , function(err) {
        FILES_FOR_COVERAGE[keep] = 1;
});
```

Данный код «проходит» по результатам регулярного выражения, запрашивая Node.js, где находится тот или иной файл, и запуская утилиту покрытия кода YUI.

Последнее, что мы должны сделать, — выполнить тесты на инструментированных версиях файлов и сохранить полученные результаты о покрытии кода. При этом переменная `yuitest_coverage` является объектом JavaScript, который должен быть преобразован в формат JSON и сохранен. В завершение его можно преобразовать в формат LCOV, и по этому формату может быть сгенерирован симпатичный HTML-код. На этом можно считать, что наша задача выполнена:

```
var coverOutFile = 'cover.json';
fs.writeFileSync(coverOutFile, JSON.stringify(_yuitest_coverage));

exec([ 'java', '-jar', coverageReportJar, '--format', 'lcov', '-o'

  , dirname, coverOutFile ].join(' '), function(err, stdout, stderr) {


});
```

В принципе, есть много разных способов получения информации о покрытии кода с использованием Node.js. Для широты кругозора рассмотрим еще один, основанный на использовании расширений. По умолчанию загрузчик Node.js «понимает» три вида расширений файлов — .js, .json и .node. При поиске загрузчиком Node.js файла для загрузки будут использоваться эти расширения, если расширение файла не было предоставлено явно. Из-за синхронной природы загрузчика мы, к сожалению, не можем динамически генерировать информацию о покрытии во время загрузки, таким образом, мы все еще нуждаемся в приеме `require ('модуль', true)`, чтобы опреде-

лить, для каких файлов нужно предварительно создать файлы покрытия. Напомню, что инструкция `require` используется для указания, нужно ли для модуля, заданного в первом параметре, создавать файл покрытия. Если второй параметр равен `true`, будет создан файл покрытия.

Если второй параметр `require` не задан, тогда инструкция require используется для загрузки файла покрытия. Мы можем указать полный путь к файлу покрытия, но чтобы было понятнее, что мы загружаем именно файл покрытия, мы рекомендуем для таких файлов использовать расширение .cover.

Также нам нужно вывести сгенерированный файл покрытия в тот же каталог, что и исходный файл, и назначить ему новое расширение, чтобы исходный загрузчик загрузил его для нас. Давайте сделаем это. Для начала заглянем в наш тестовый сценарий:

```
require('./src/myModule.cover');
```

Этот оператор загрузит файл покрытия для нашего файла (то есть для файла модуля myModule.js используется файл покрытия myModule.cover). Регулярное выражение в свою очередь будет изменено так:

```
/require\s*\(\s*['"]([^'"]+)\.cover['"]\)/g
```

Как только что было сказано, мы договорились хранить вместе исходные файлы и их файлы покрытия. Поэтому при создании файла покрытия вместо его дампа в `coverageBase` (обычно это каталог `/tmp`) мы просто помещаем его в каталог, в котором находится исходный файл, но используем расширение `.cover`:

```
var realFile = require.resolve(file)
        , coverFile = realFile.replace('.js', '.cover');
;
exec('java -jar ' + coverageJar + " -o " + coverFile + " " + realFile,
  function(err) {});
```

Нам больше не нужно отслеживать, для каких файлов созданы файлы покрытия, поскольку загрузчик загрузит нужный файл в зависимости от расширения.

Наконец, мы говорим загрузчику, что он должен сделать, когда встретит файл с расширением `.cover`:

```
require.extensions['.cover'] = require.extensions['.js'];
```

Это то, что загрузчик должен сделать с файлами с расширением `.js`. Есть несколько способов сгенерировать информацию о покрытия кода для серверных JavaScript-сценариев. Выберите тот, который больше вам подходит. Если приведенные здесь решения вам не понравились, обратитесь к главе 8, в которой будет рассмотрено автоматическое создание информации покрытия.

5.5. Сохранение информации о покрытии

Сохранить информацию о покрытии, это значит сначала извлечь ее из памяти браузера, а затем сохранить локально на диске. Слово «локально» означает размещение на веб-сервере, там же, где хранятся файлы тестов.

Помните, что каждое обновление страницы сбрасывает информацию о покрытии JavaScript-кода, загруженного на странице, поэтому прежде, чем обновлять страницу, нужно сохранить информацию о покрытии где-нибудь на диске, иначе она будет потеряна навсегда.

В случае модульного тестирования

Для модульного тестирования сохранение информации о покрытии кода представляет ту же проблему, что и сохранение результатов тестов. Для сохранения нам нужно отправить эту информацию обратно на сервер, используя или метод POST, или Ajax. И если обычно при тестировании можно обойтись и без веб-сервера (просто загружаете связующий HTML-файл в свой браузер, и тесты будут запущены), то для сохранения результатов тестирования и информации о покрытии вы без помощи веб-сервера не обойдетесь.

Фреймворк YUI Test предоставляет различные вспомогательные средства для получения этой информации в разных форматах:

```
var TestRunner = Y.Test.Runner;
TestRunner.subscribe(TestRunner.TEST_SUITE_COMPLETE_EVENT, getResults);
TestRunner.run();
function getResults(data) {
```

```
  var reporter = new Y.Test.Reporter(
    "http://www.yourserver.com/path/to/target",
    Y.Test.Format.JUnitXML);
  reporter.report(results);
}
```

Добавьте этот фрагмент кода в конец каждого сценария тестирования, и отчеты о тестировании будут отправлены на ваш сервер в формате XML JUnit, который распознается Hudson/Jenkins и многими другими инструментами сборки. YUI также поддерживает вывод в форматах XML, JSON и TAP. С форматами JSON и XML вы уже знакомы, а формат TAP пришел к нам из мира Perl и также распознается всеми основными инструментами сборки.

Рассмотрим следующий код:

```
function getResults(data) {
    // Используйте формат JUnitXML для результатов тестирования модулей
    var reporter = new Y.Test.Reporter(
      "http://www.yourserver.com/path/to/target",
      Y.Test.Format.JUnitXML);
    // Получаем и кодируем результаты покрытия
    reporter.addField("coverageResults",
      Y.Test.Runner.getCoverage(Y.Coverage.Format.JSON));
    // Отправляем их
    reporter.report(results);
}
```

Метод `addField` объекта `reporter` позволяет вам передавать произвольные данные обратно на сервер. Таким образом, получается, что мы получаем результаты определения покрытия кода, кодируем эти результаты как JSON и передаем их обратно вместе с результатами модульного тестирования. Обратите внимание на то, что JSON – по большому счету единственный нормальный формат для упаковки информации о покрытии (несмотря на наличие альтернатив); и, кстати, именно в этом формате ожидает данные инструмент создания отчетов о покрытии во фреймворке YUI.

Конечный результат отправляется на наш сервер методом POST и содержит в себе значения параметров `results` и `coverageResults`. Соответственно на стороне сервера, после получения, эти два фрагмента данных могут быть сохранены в локальной файловой системе.

В случае интеграционного тестирования

Процесс сохранения информации о покрытии интеграционного тестирования с использованием Selenium аналогичен. После выполнения тестов Selenium, но перед тем, как уйти с текущей страницы, нужно получить информацию о покрытии и передать ее обратно на сервер. Selenium предоставляет функции `tearDown` и `after`, в которые и нужно добавить код для обеспечения захвата результатов покрытия.

Лучше всего в Selenium это делать с использованием глобальной переменной `_yuitest_coverage`, в которую помещается вся информация о покрытии. В Selenium1 код получения информации о покрытии будет выглядеть так:

```
String coverage = selenium.getEval(
  "JSON.stringify(window._yuitest_coverage)");
```

При использовании Selenium2/WebDriver код будет следующим:

```
String coverage = (String)((JavaScriptExecutor) driver)
  .executeScript("return JSON.stringify(window._yuitest_coverage);");
```

Затем просто выведите переменную `coverage` в файл. В идеале, этот файл должен называться так же, как и ваш тест, например `testClick.coverage.json`.

Имейте в виду, что, если ваше приложение использует фреймы (iframes), информация о покрытии кода из iframe не будет получена посредством высокоуровневой переменной `yuitest_coverage`. После захвата покрытия кода из высокоуровневого главного окна вам нужно указать Selenium1, какой фрейм использовать:

```
selenium.SelectFrame("src=foo.html");
```

В Selenium2-3/WebDriver нужно использовать следующий код:

```
driver.SwitchTo().Frame(driver.FindElement(By.CssSelector(
    "iframe[src=\"foo.html\"]")));
```

Кроме того, можно просто использовать селектор Selenium для выбора нужного фрейма, для которого нужно получить информацию о покрытии.

В случае с фреймами обязательно агрегируйте значения `_yuitest_coverage` для каждого фрейма в общий отчет покрытия. О том, как это сделать, написано в разделе «Агрегация» далее в этой главе.

5.6. Генерация вывода

Формат JUnit XML «понимают» большинство (если не все) инструментов сборки, чего не скажешь о формате покрытия YUI JSON. К счастью, довольно просто конвертировать вывод в формате YUI JSON в стандарт LCOV, который понимают большинство утилит сборки. Для этого введите следующую команду (все в одной строке):

```
% java -jar yuitest-coverage-report.jar --format LCOV
  -o <output_directory> coverage.json
```

Здесь `coverage.json` — это JSON-файл, полученный из POST-параметра `coverageResults` (или непосредственно из переменной `_yuitest_coverage` интеграционного тестирования). Результат выполнения этой команды (информация о покрытии в формате LCOV) будет помещен в каталог `<output_directory>`. Полученный файл можно передать для дальнейшей обработки утилитам `lcov` и `genhtml`. Hudson/Jenkins поддерживает этот формат "из коробки" и может сгенерировать HTML-файл с информацией о покрытии. Также для создания красиво оформленного HTML-кода можно использовать команду `genhtml`. Подробная информация о `lcon`, `genhtml` и им подобным утилитам доступна по адресу: http://ltp.sourceforge.net/coverage/lcov.php.

Пример вывода `gethtml` приведен здесь: http://ltp.sourceforge.net/coverage/lcov/output/index.html. При этом для тех, кто не доволен видом по умолчанию, отметим, что команда `genhtml` принимает множество параметров, позволяющих настраивать HTML-вывод.

Рисунок 5.1 демонстрирует, на что будет похож вывод, а рисунок 5.2 показывает наш файл, содержащий функцию sum. Было покрыто две из двух строк одной функции.

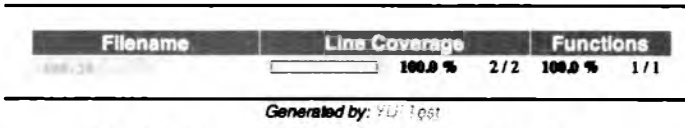


Рис. 5.1. Результаты покрытия кода для sum.js

```
    Branch data     Line data    Source code
1                   1            : function sum(a ,b) {
2                   1            :  return a + b;
3                   :            :  }
4                   :            :
```

Рис. 5.2. Покрытие кода для sum.js

Мы видим, что каждая строка выполнялась один раз, что дает нам 100%-ое покрытие кода.

5.7. Агрегация

В случае с модульным и интеграционным тестированием мы смогли сохранить информацию о покрытии для отдельных тестов или наборов тестов. Но как увидеть полную картину? Для этого на помощь приходит команда `lcov`.

Команда `lcov -a` агрегирует множество файлов в формате LCOV в один большой файл. Так, следующая команда:

```
% lcov -a test.lcov -a test1.lcov -a test2.lcov ... -o total.lcov
```

объединяет набор LCOV-файлов. Причем данная команда может использоваться для соединения всех тестов как модульного, так и интеграционного тестирования вместе.

Чтобы быть объединенными, все LCOV-файлы должны совместно использовать один и тот же корневой каталог. Это не является проблемой при объединении всех результатов покрытия модульного тестирования или всех результатов покрытия интеграционного тестирования независимо друг от друга, но может стать большой проблемой, когда нужно объединить вместе результаты покрытия модульного и интеграционного тестирований.

Посмотрите на формат LCOV. Первая строка каждого нового раздела файла покрытия начинается с полного пути к исходному файлу (в нашем случае к JavaScript-файлу). Если встречаются *два несовпадающих корневых каталога*, `lcov` будет не в состоянии объединить их.

Таким образом, нам нужен простой shell-сценарий (сценарий оболочки), гарантирующий, что корневые каталоги будут одинаковыми (у машины, где происходит слияние, должно быть исходное корневое дерево). Наиболее вероятно, что один из наборов корневых каталогов будет правильным, а другой — нет. Поскольку LCOV-файлы — это обычные текстовые файлы, дело сводится к изменению корневого каталога некорректного LCOV-файла: новый каталог должен указывать на расположение, в котором фактически находится код.

Если ваш исходный код находится в каталоге `/a/b/c`, но в LCOV-файле интеграционного тестирования указано, что исходный код находится в каталоге `/d/e/f`, просто напишите на любимом языке сценарий, преобразовывающий строку `/d/e/f` в строку `/a/b/c`. На Perl сценарий будет таким:

```
my $old = '/a/b/c';
my $new = '/d/e/f';
open(F, "wrong.lcov");
open(G, ">right.lcov");
while(<F>) {
        s#^$old#$new#;
        print G;
}
close(G);
close(F);
```

В приведенном коде `wrong.lcov` конвертируется так, чтобы быть соединенным с другим LCOV-файлом, находящимся в каталоге `/d/e/f`. Как только у вас будет итоговый файл `total.lcov`, получить привлекательный HTML-код очень просто:

```
% genhtml -o /path/to/docRoot/coverage total.lcov
```

Теперь направьте ваш браузер в каталог `/coverage` на вашем сервере и посмотрите общий отчет о покрытии.

Сгенерированный вывод — или одиночный файл, или агрегированный LCOV-файл — может быть передан в автоматизированный инструмент сборки, такой как Hudson/Jenkins. Старые версии Hudson (до 2011 г.) понимают LCOV-отформатированные файлы, и все, что вам нужно сделать для работы с ними в старых версиях Hudson, – это указать, где находится ваш агрегированный файл, который бу-

дет включен в вывод сборки. Рисунок 5.3 показывает, как это сконфигурировать.

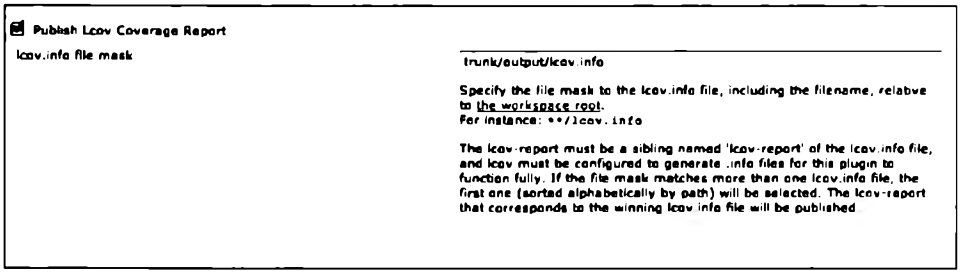


Рис. 5.3. Экран настройки Hudson

В результате Hudson обрабатывает HTML-файлы и сохраняет данные о покрытии через разные сборки, чтобы было легко отследить покрытие кода в течение долгого времени. На главной странице вы найдете кнопку, показанную на рис. 5.4.

Рис. 5.4. Кнопка просмотра отчета lcov

Нажмите эту кнопку, и вы увидите таблицы со значениями метрики покрытия кода (рис. 5.4).

К сожалению, у нынешних версий Jenkins (современная версия Hudson, которая в 2011 году получила свое новое название в ответ на то, что компания Oracle отказалась «отдать» права на торговую марку Hudson) нет такой функциональности. Как быть в таких ситуациях, мы узнаем в главе 8.

5.8. Скрытые файлы

Что насчет файлов, которые есть в нашем проекте, но для которых нет данных о покрытии кода, для них и тестов-то может еще не быть? Эти файлы не попадают в общий отчет и никак не учитываются. Однако для того, чтобы иметь более адекватную картину о

покрытии кода нашими имеющимися тестами, мы должны учесть и эти файлы, код которых пройдет как непокрытый и снизит общее агрегированное значение метрики покрытия кода.

Самый простой метод учесть эти скрытые файлы – сделать так, чтобы для *всех* файлов вашего приложения были созданы тестовые сценарии (тестовые файлы), просто для некоторых файлов тестовые сценарии будут «пустыми». При запуске пустого теста на инструментированном файле со счетчиками покрытия будет сгенерирован вывод LCOV (0%), что означает нулевое покрытие. Теперь можно сохранить и обработать эти LCOV-данные, как обычно, и при этом весь ваш код будет учтен в агрегированной сводке покрытия.

Далее приведен HTML пустого теста, который должны иметь все файлы приложения. Как только будет готов соответствующий непустой тестовый сценарий, вы можете удалить этот HTML-файл и заменить его реальным тестом:

```
<html lang="en">
<body class="yui3-skin-sam">
  <div id="log"></div>
  <h3>Пустой тест для APP_FILE</h3>
  <script src="http://yui.yahooapis.com/3.18.1/build/yui/yui.js">
  </script>
  <script src="/path/to/coveraged/file/without/tests.js"></script>
  <script>
    YUI().use('test', function(Y) {
      Y.Test.Runner.add(
        new Y.Test.Suite('no test').add(
          new Y.Test.Case({
            name: 'dummy_NOTESTS'
            , 'testFile': function() {
              Y.Assert.areEqual(true, true);
            }
          })
        )
      );
      Y.TestRunner.go();
    });
  </script>
</body>
</html>
```

Такого пустого теста достаточно, чтобы загрузить инструментированный файл со счетчиками покрытия и получить нулевую информацию о покрытии, сохраненную со всеми остальными значениями метрик покрытия кода в агрегированном отчете. При этом при изучении общего отчета будет сразу понятно, у каких файлов нет тестов, покрывающих их.

Кстати говоря, в «больших» средах разработки эти пустые тестовые файлы должны быть сгенерированы автоматически по итогам сравнения связующего HTML-кода со всем кодом вашего приложения и определения, где есть «пробелы». Автоматически сгенерированная версия этого файла будет создаваться динамически и включаться в ваши проходы теста.

Лучший способ найти «скрытые» файлы (файлы, для которых нет тестовых сценариев и которые поэтому не видны) — создать список всех файлов приложения и список файлов тестов, а затем сравнить эти два списка. Для этого можно использовать следующий Perl-сценарий (приведенную ниже команду нужно ввести в одной строке):

```
% perl find_no_unit_tests.pl --test_dir test --src_dir dir1
--src_dir dir2 ... --src_base /homes/trostler/mycoolapp
```

Данный сценарий «проходит» по корневому каталогу с тестами (в данном случае он называется *test*) и по корневому каталогу исходного кода, в котором и находятся JavaScript-файлы. Опция `src_base` задает корневой каталог вашего проекта. Этот сценарий создает список всех файлов с исходным JavaScript-кодом и список всех JavaScript-файлов тестов, а затем выводит разницу между этими двумя списками:

```
#!/usr/local/bin/perl
use Getopt::Long;
use File::Find;
use File::Basename;
my($debug, $test_dir, @src_dir, $src_base);
my $src_key = 'src'; // корневой каталог дерева исходного кода
GetOptions (
        "test_dir=s" => \$test_dir,
        "src_dir=s" => \@src_dir,
        "src_base=s" => \$src_base,
```

```
    ) || die "Неправильные параметры!\n";;
    my $src_files = {};
    find(\&all_src_files, @src_dir);
    find(\&all_tested_files, $test_dir);
```

Мы принимаем параметры командной строки, "проходимся" по всем каталогам с исходным кодом и извлекаем имена всех JavaScript-файлов:

```
    sub all_src_files {
      return unless (/\.js$/);
      foreach my $src_dir (@src_dir) {
        $File::Find::name =~ s/^\Q$src_base\E//;
      }
      $src_files->{$File::Find::name}++;
    }
```

Теперь хэш `%src_files` содержит все ваши JavaScript-файлы с исходным кодом. Далее приведен код для «прохода» по файлам тестов:

```
sub all_tested_files {
    return unless (/\.html?$/);
    open(F, $_) || die "Can't open $_: $!\n";
    while(my $line = <F>) {
        if ($line =~ /["']([^"]+?\/($src_key\/[^"]+?\.js))["']/) {
            my($full_file_path) = $2;
            print "Файл теста $File::Find::name покрыт
                $full_file_path\n" if ($debug);
            delete $src_files->{$full_file_path};
        }
    }
}
```

Самой неприятной вещью здесь, безусловно, является regex-поиск тегов сценария вида:

```
<script src="/path/to/JS/file"></script>
```

Как только имя файла будет удалено из хэша `%src_files`, оно будет помечено как «покрытое».

После этого хэш `%src_files` содержит только JavaScript-файлы без соответствующих тестов. Теперь очень просто использовать какую-либо систему шаблонов для создания пустого теста для каждого из

этих файлов. Причем, вы можете сохранить эти пустые тесты где-то в вашем каталоге *tests*, чтобы они могли быть запущены позже вашей автоматизированной системой запуска тестов (одну из таких систем мы рассмотрим в главе 8).

5.9. Цели покрытия

Как правило, цели определения значения метрики покрытия кода для модульного тестирования отличаются от целей определения метрики покрытия для интеграционного тестирования. Поскольку интеграционные тесты покрывают существенно больше кода, сложнее определить корреляцию между тем, что было покрыто и что было протестировано. Это — проявление той же самой проблемы, замеченной с покрытием кода и модульным тестированием: то, что строка кода выполнена тестом (была покрыта), еще не означает, что код «протестирован», что он корректен.

В случае с модульным тестированием значение покрытия кода может использоваться в ходе анализа корректности функционирования, полноты тестирования того или иного метода, тогда как в случае с интеграционным тестированием такие цели нереальны.

В случае с интеграционным тестированием более реальными целями является определение местонахождения той или иной этой функции в исходном коде и насколько часто она вызывается/используется.

Это позволяет разработчикам и специалистам по обеспечению качества определить части системы, которые, при нормальной работе, используются очень редко или никогда не используются (такие как код обработки ошибок и т.п.), а также позволяет сориентировать тестировщиков на тестирование наиболее важных режимов, сконцентрировать свои тесты на наиболее часто используемых функциях.

К какому значению покрытия кода следует стремиться? Какие значения покрытия кода являются хорошими, а какие – плохими? На это и не может быть однозначного ответа, все зависит от контекста. Одно ясно — эти числа должны увеличиться со временем.

В большинстве случаев нормальной практикой, которой я придерживаюсь, является стремление к 80%-ному покрытию строк кода. Особенно при модульном тестировании. В любом случае метрики покрытия кода должны быть вторичны по отношению к тестированию, а не наоборот. Технически довольно просто получить большое значение покрытия кода при абсолютно негодном коде.

Резюме

Определение и мониторинг информации о покрытии кода крайне важны для модульного тестирования и важны для интеграционного тестирования. И хотя значения метрики покрытия кода не раскрывают нам всей картины об эффективности тестирования, они по сути являются единственными числовыми характеристиками, численными показателями для отслеживания процесса тестирования.

Вы, ваш босс и кто-либо еще может сразу увидеть, сколько кода покрыто тестами. Небольшие проценты покрытия указывают те участки, на которых нужно сфокусировать будущие усилия по тестированию.

Можно объединять результаты тестирования разных модулей, а также результаты покрытия кода модульным и интеграционным тестированием, с целью получения единого отчета. В главе 8 мы обсудим, как автоматизировать этот процесс, используя окружение JavaScript Unit Test Environment.

Наряду со статическим анализом кода отслеживание покрытия кода является отличной метрикой анализа вашего кода. При этом необходимо понимать, что никакое одно число не может дать полную картинку вашего кода или ваших тестов, но сбор и отслеживание этих чисел в течение долгого времени обеспечивают понимание того, как развивается ваш код.

Высокое значение покрытия кода не должно вводить в заблуждение относительно качества тестирования: если строка кода покрыта (выполнена), это еще не значит, что она сделала это корректно. Достижение высоких значений покрытия кода должно быть побочным продуктом хорошего тестирования, но не самой целью. Помните, что вы пишете все эти тесты не для получения более высоких метрик, а чтобы убедиться, что ваш код корректен и надежен.

Глава 6.

Интеграционное, нагрузочное и тестирование производительности

В дополнение к модульному тестированию для достижения «хорошести» кода также важно произвести интеграционное, нагрузочное тестирование и тестирование производительности вашего приложения. Написание интеграционных тестов разных видов (для тестирования в «реальных» браузерах или для тестирования в тестировочной среде) довольно просто, и в данной главе мы посмотрим, как это делается.

Также мы рассмотрим создание нагрузочных тестов, для чего сгенерируем стандартный водопадный график времени загрузки приложения. Все это тоже несложно.

6.1. Важность интеграции

Все части приложения в итоге должны быть объединены вместе — либо в среде для тестирования, либо в производственной среде. Как мы уже знаем, хороший JavaScript-код предполагает разбиение на небольшие части с минимальными зависимостями. В процессе интеграционного тестирования все эти части будут объединены между собой для группового тестирования.

Событийно-ориентированная архитектура — пример большого числа слабо связанных частей, которые должны работать слаженно, как единый механизм, а пока мы не протестируем эту слаженную работу, говорить об идеальности кода еще рано.

Сразу отметим, что для интеграционного тестирования очень важно наличие автоматизации, которая бы развернула систему и перевела бы ее в рабочее состояние. Как только это произошло, можно приступить к интеграционному тестированию.

6.2. Интеграционное тестирование. Selenium

Проведение интеграционного теста веб-приложения подразумевает запуск вашего приложения в браузере и гарантирует, что приложение работает так, как и ожидалось. Тестирование отдельных частей приложения посредством модульного тестирования — это хорошо, но также важно убедиться, что все части работают вместе надлежащим образом. Для этого и предназначено интеграционное тестирование. На этом уровне тестирования нет никаких имитаций или заглушек для тестирования зависимостей, так как тестирование производится на уровне всего приложения.

Серия продуктов Selenium

Тестирование JavaScript-кода в браузере обычно подразумевает использование Selenium. Изначально, Selenium – это инструмент для автоматизированного управления браузерами. Однако самой востребованной областью применения Selenium является автоматизация тестирования веб-приложений. При этом Selenium поддерживает работу со всеми основными браузерами (Firefox, Internet Explorer, Safari, Opera, Chrome) под все основные операционные системы (Microsoft Windows, Mac OS, Linux).

По сути дела Selenium – это не одна отдельная программа, а проект, в рамках которого разрабатывается целая серия программных продуктов с открытым исходным кодом: Selenium IDE, Selenium WebDriver, Selenium RC, Selenium Server, Selenium Grid. При этом Selenium WebDriver еще называют Selenium 2 или 3.

Какой из них вам нужен? Это зависит от ваших задач:

- Если вам требуется «быстрое» решение, небольшой сценарий для быстрого автоматизированного воспроизведения бага или вспомогательный скрипт для выполнения отдельных рутинных действий при ручном тестировании, то вам следует обратиться к **Selenium IDE** – расширению браузера Firefox, которое позволяет записывать и воспроизводить действия пользователя в браузере.

- В том случае, если вам требуется разработать надежный фреймворк автоматизации, способный работать с любым браузером, или у вас большой тестовый набор, включающий тесты с достаточно сложной логикой поведения и проверок, то вам нужен **Selenium WebDriver (Selenium 3 или просто WebDriver).** Selenium WebDriver – большая программная библиотека, содержащая в себе целое семейство драйверов для различных браузеров, а также набор клиентских библиотек на разных языках, позволяющих работать с этими драйверами (управлять браузером из программы, написанной на том или ином языке программирования). WebDriver не является инструментом тестирования, но предоставляет автотестам доступ к браузеру.

- **Selenium RC** – это предшественник Selenium WebDriver, развитие которого на данный момент «заморожено».

- В том случае, если вам потребуется запускать тесты удаленно на разных машинах с разными операционными системами и браузерами и/или реализовать тестовый стенд для выполнения большого количества тестов, то вам потребуется использовать **Selenium Server**, который позволяет управлять браузером с удаленной машины по сети: принимать команды с удаленной машины, где работает сценарий автоматизации, и исполнять их в браузере на машине с установленным сервером. Из нескольких серверов Selenium вы можете создать распределенную сеть тестирования, которая называется Selenium Grid и позволяет легко масштабировать тестовый стенд.

Далее мы рассмотрим, как на практике реализовать решение тех или иных задач тестирования с использованием Selenium.

Как уже было сказано, Selenium IDE – самый быстрый способ что-либо протестировать. Перейдите на сайт SeleniumHQ (http://www.seleniumhq.org/download) и загрузите последнюю версию IDE (2.9.0 на момент написания этих строк). Предварительно у вас должен быть установлен браузер Firefox, в настройках которого должна быть разрешена установка расширений (дополнений).

Откройте ваш веб-сайт в Firefox'е и откройте Selenium IDE (**Tools > Selenium IDE**). Установите в параметр **Base URL** адрес вашей страницы, по которому запускается ваше веб-приложение. Нажмите кнопку записи в верхнем правом углу Selenium IDE, и Selenium начнет отслеживать перемещения мыши и нажатия клавиш клавиатуры при работе с вашим приложением в браузере Firefox. Для окончания отслеживания снова нажмите эту кнопку.

Выберите команду меню **File > Export Test Case As** для сохранения ваших перемещений, щелчков мыши, нажатий клавиш (в общем, тестового случая) на одном из поддерживаемых Selenium2/WebDriver языков или на оригинальном языке Selenium (Remote Control), который использовать не нужно, если вы плохо знакомы с Selenium.

Для повторного воспроизведения этих тестов в Selenium IDE нажмите зеленую кнопку воспроизведения. Журнал внизу окна Selenium IDE позволит вам наблюдать, что происходит.

При работе с Selenium есть одна проблема: по умолчанию эта среда использует ID элемента для идентификации элементов, с которыми вы взаимодействуете. Проблема происходит при использовании JavaScript-фреймворков, которые динамически генерируют идентификаторы элементов, — ваши тесты перестанут работать, поскольку при динамической генерации ID элемента сменится при следующем выполнении и не будет найден. К счастью, текстовое поле **Target** в Selenium IDE позволяет вам исправлять эту ситуацию с помощью XPath или CSS-выражений для определения местоположения элементов вместо использования ID. Нажмите кнопку поиска (она находится возле поля **Target**) для определения местоположения элементов, на которых нужно "сфокусироваться" при изменении селекторов. Ну и конечно, если вы установите идентификаторы элементов вручную, у вас тоже не будет этой проблемы.

Впоследствии вы можете запустить сохраненные тестовые случаи из командной строки с использованием JUnit[1]. Для этого просто экспортируйте ваш тестовый случай как JUnit 5 (WebDriver BackEnd). При этом в начало файла IDE добавит следующее объявление:

1 JUnit — библиотека для модульного тестирования программного обеспечения на языке Java.

```
package com.example.tests;
```

Измените это объявление так, чтобы оно соответствовало вашему окружению (вместо example укажите имя вашего класса) или просто удалите эту строку.

Для дальнейших опытов нам понадобятся установленные сервер Selenium Server и клиентские Java-драйверы. Загрузите текущую версию сервера Selenium Server (3.141.59 на момент написания этих строк) и Java Selenium client driver (3.141.59 на момент написания этих строк) – WebDriver под Java – с сайта **http://www.seleniumhq.org/download/**. Вам нужно распаковать клиент Java Selenium.

Теперь откомпилируем экспортированный нами ранее Selenium-сценарий с помощью `selenium-server.jar`:

```
% java -cp path/to/selenium-server-standalone-3.141.59.jar test.java
```

Данная команда откомпилирует экспортируемый вами тестовый случай Selenium. Чтобы выполнить ваш тест, вам сначала нужно запустить сервер Selenium:

```
% java -jar path/to/selenium-server-standalone-3.141.59.jar
```

Ну а затем «скормить» ваш тестовый JUnit-сценарий (введите все в одной строке):

```
% java -cp path/to/selenium-server-standalone-3.141.59.jar:Downloads/
selenium-2.20.0/libs/junit-dep-5.1.jar:.
org.junit.runner.JUnitCore test
```

Понятно, что вы должны указать свои пути к файлу сервера Selenium (`selenium-server-standalone-3.141.59.jar`) и к JAR-файлу JUnit (`junit-dep-5.1.jar`).

Приведенная ранее команда подразумевает, что вы удалили объявление `package`. Если нет, вам нужно использовать немного другую команду (опять все в одной строке):

```
% java -cp selenium-server-standalone-3.141.59.jar
:selenium-3.141.59/libs/junit-dep-5.1.jar:.
org.junit.runner.JUnitCore com.example.tests.test
```

Откомпилированная Java-программа должна находиться в `com/examples/test`, чтобы интерпретатор Java смог найти ее (или вы можете изменить путь класса).

Использование Selenium IDE — дело не всегда благодарное, а автоматически созданные тесты довольно часто неуклюжи. Гораздо лучше создавать тестовые сценарии вручную (для этого вы можете использовать JUnit). Однако использование средств Selenium IDE полезно в «общеобразовательных» целях, когда вы сможете изучить XPath, селекторы CSS, узнать, как правильно использовать атрибуты ID в вашем HTML-коде, чтобы Selenium мог «захватить» их и управлять вашим приложением путем нажатия на ссылки и кнопки, перетаскивания элементов, редактирования элементов формы и т.д.

На заметку

Для проверки функциональности вашего приложения вы можете использовать семейства Selenium-функций `assert` и `verify`. Заметьте, что семейство `assert` в случае сбоя немедленно перейдет к следующей тестовой функции, а семейство `verify` продолжит выполнение дальнейшего кода внутри тестовой функции, а сообщение об ошибке будет внесено в специальный список. Функции семейства `verify` можно использовать для некритичных тестов, после которых можно продолжать выполнение, даже если тест дал отрицательный результат. Практически всегда в тестах Selenium нужно использовать функции типа `assert`.

WebDriverjs и написание интеграционных тестов на JavaScript

Очень важно, что существует JavaScript-версия WebDriver, носящая незамысловатое название **webdriverjs** и позволяющая писать интеграционные тесты Selenium, используя наш любимый JavaScript. Все это реализовано в виде npm-пакета для Node.JS.

Использование npm-модуля **webdriverjs** (https://github.com/Camme/webdriverjs) в качестве драйвера очень простое:

```
var webdriverjs = require("webdriverjs")
  , browser = webdriverjs.remote({
    host: 'localhost'
    , port: 4444
    , desiredCapabilities: { browserName: 'firefox' }
  })
```

```
;
browser
   .testMode()
   .init()
    .url("http://search.yahoo.com")
    .setValue("#yschsp", "JavaScript")
    .submitForm("#sf")
    .tests.visible('#resultCount', true, 'Got result count')
    .end();
```

На фоне запущенного локально сервера Selenium Server вышеприведенный код запустит браузер Firefox и откроет в нем страницу поисковика Yahoo (http://search.yahoo.com) с поисковым запросом «JavaScript», и обеспечит, что элемент с `id` равным `resultCount` будет видимым.

Создать скриншот средствами Selenium очень легко. Просто добавьте вызов `saveScreenshot`:

```
var webdriverjs = require("webdriverjs")
   , browser = webdriverjs.remote({
       host: 'localhost'
       , port: 4444
       , desiredCapabilities: { browserName: 'firefox' }
   })
;
browser
   .testMode()
   .init()
   .url("http://search.yahoo.com")
   .setValue("#yschsp", "javascript")
   .submitForm("#sf")
   .tests.visible('#resultCount', true, 'Got result count')
   .saveScreenshot('results.png')
   .end();
```

И теперь у вас есть прекрасный скриншот, показанный на рис. 6.1. Обратите внимание, что реклама Yahoo! Axis отображается посередине рис. 6.1. По идее она должна быть внизу *видимой области* страницы, но Selenium делает снимок всей страницы, независимо от того, поместилась она целиком на экране или нет. Поэтому и получилось, что в данном случае видимая область закончилась посере-

Рис. 6.1. Создание снимка экрана с помощью Selenium

дине страницы, а внизу идет продолжение страницы, которое, так сказать, "не видно". При реальном просмотре страницы в браузере баннер прилеплен к низу видимой области и остается там при прокрутке содержимого страницы.

Это мы работали под Firefox, который используется по умолчанию. Чтобы запустить этот же пример в Chrome, вам нужно загрузить драйвер Chrome для вашей операционной системы (https://code.google.com/p/selenium/wiki/ChromeDriver) и установить его в любой из каталогов, определенный в переменной окружения PATH. Затем просто измените эту строчку:

```
, desiredCapabilities: { browserName: 'firefox' }
```

исправив на:

```
, desiredCapabilities: { browserName: 'chrome' }
```

Так, а что насчет Internet Explorer'а? Для него тоже есть свой драйвер. Последний для вашей платформы драйвер IE можно загрузить с сайта **https://code.google.com/p/selenium/downloads/list**. Затем поместите исполнимый файл в один из каталогов, указанных в переменной PATH, и запустите его. По умолчанию драйвер использует порт 5555:

```
, port: 5555
, desiredCapabilities: { browserName: 'internetExplorer' }
```

Selenium Remote Control

Программное решение Selenium Remote Control является «замороженной» веткой развития Selenium, место которой занял поддерживаемый и развивающийся WebDriver. Однако могут быть ситуации, когда вам может пригодиться Selenium RT.

Дело в том, что WebDriver не является результатом эволюционного развития Selemium RT и оба этих продукта построены на очень разных принципах, и у них практически нет общего кода. Так вот, Selenium RT при создании тестовых сценариев использует свой скриптовый язык, условно называемый Selenium 1, который по своей сути является JavaScript. Тогда как WebDriver использует уже свой индивидуальный язык Selenium.

Вторая и третья версия, конечно же, лучше, функциональнее и позволяет более эффективно управлять браузерами, но достоинство Selenium 1, из-за которого еще не забыли окончательно, состоит в том, что его в принципе поддерживает большее количество браузеров, так как практически каждый браузер может понять и исполнить JavaScript-код. Вот в таком контексте вам и может пригодиться Selemium RT.

Аналогом **webdriverjs** под Selenium Remote Control является отличный npm-модуль **soda** (https://github.com/LearnBoost/soda). И ниже приведен тот же пример, что и раньше, но с использованием модуля **soda** и браузера Safari:

```
var soda = require('soda')
  , browser = soda.createClient({
    url: 'http://search.yahoo.com'
    , host: 'localhost'
    , browser: 'safari'
  })
;
browser
  .chain
  .session()
  .open('/')
  .type('yschsp', 'JavaScript')
  .submit('sf')
  .waitForPageToLoad(5000)
  .assertElementPresent('resultCount')
  .end(function(err) {
   browser.testComplete(function() {
    if (err) {
      console.log('Test failures: ' + err);
    } else {
      console.log('success!');
     }
   })
});
```

Модуль **soda** подобен модулю **webdriverjs**, но теперь вместо Selenium-команд вы используете Selenium1-команды (http://release.seleniumhq.org/selenium-core/1.0.1/reference.html).

Обратите внимание, что поменялись только клиентские команды, автономный сервер Selenium (`selenium-server-standalone-2.37.0.jar`) понимает как команды Selenium1, так и Selenium2/3, поэтому серверная часть останется неизменной.

Selenium Grid

Selenium также поддерживает конфигурацию "сетки", включающую один центральный "концентратор" и много распределенных "узлов", которые порождают браузеры и передают им команды. Эта конфигурация отлично подходит для параллельной обработки заданий Selenium или для создания центрального репозитария, предоставляющего разработчику централизованный доступ к Selenium без необходимости запуска и обслуживания собственного экземпляра Selenium у каждого программиста.

Каждый "узел" подключается к центральному концентратору с порожденным в нем браузером, а концентратор обрабатывает задания Selenium. Один и тот же концентратор может обработать клиентов Mac/Windows/Linux, на которых выполняются различные браузеры.

Реализуется это все тем же Server Selenium. Удобно, что последняя версия сервера Server Selenium поддерживает и WebDriver, и Remote Control для сеток.

Чтобы запустить концентратор сетки, введите следующую команду:

```
% java -jar selenium-server-standalone-3.141.59.jar -role hub
```

Как только концентратор запущен, запустите узлы, которые подключатся к концентратору и породят браузеры, используя следующий код (все в одной строке) — узлы могут работать на том же компьютере, что и концентратор, или же на удаленном компьютере:

```
% java -jar selenium-server-standalone-3.141.59.jar
-role node -hub http://localhost:4444/grid/register
```

В данном случае мы предполагаем, что концентратор запущен на порту 4444 (по умолчанию) и находится на том же компьютере, что и узел.

Используя **webdriverjs**, вы можете использовать в своих интересах дополнительные узлы для параллелизации, и тогда вместо обработки одного запроса за один раз вы можете обрабатывать одновременно несколько запросов. Чтобы получить визуальное представление о том, как все работает (информацию о числе узлов, подключенных к концентратору, и о числе параллельно обрабатываемых заданий), введите в адресной строке браузера, запущенного на концентраторе, следующее: http://localhost:4444/grid/console.

Более старые Selenium-сетки, построенные на Selenium1 Remote Control, могут обрабатывать только одно задание Selenium на один узел. Более новые Selenium-сетки, реализованные на основе WebDriver, могут обрабатывать несколько заданий на узел. По умолчанию они обрабатывают 5 заданий, но вы можете изменить это число, используя параметр командной строки `-maxSession <число>` для каждого узла.

Таким образом, у вас получается и вы можете использовать тестировочный стенд для поддержки всего вашего тестирования, независимо от языка, на котором написаны тесты, и независимо от используемых браузеров.

6.3. CasperJS

Selenium — не единственная платформа интеграционного тестирования. Созданный на основании PhantomJS, фреймворк CasperJS обеспечивает подобную Selenium функциональность, но в абсолютно бездисплейной среде (среде, независимой от устройства отображения). Используя чистый JavaScript или CoffeeScript, вы можете написать сценарий взаимодействия со своим веб-приложением и протестировать результаты, в том числе снимки экрана, без любого Java.

При использовании CasperJS с последней версией PhantomJS (1.9.2 и 2.1.1 на данный момент) вы больше не нуждаетесь в библиотеке X11 или Xvfb для запуска браузера PhantomJS WebKit, поскольку PhantomJS теперь основан: 2.0 на библиотеке Qt5, а 1.9.x на библиотеке Lighthouse Qt, которая является независимой от устройства отображения (дисплея). Это означает, что теперь на ваших серверах возможно действительно бездисплейное интеграционное тестирование.

Конечно, если вы захотите протестировать ваш код в Internet Explorer, вам придется использовать Selenium. Но CasperJS — хорошее дополнение в вашем наборе инструментов «полировки» вашего кода для быстрого тестирования в бездисплейной среде.

Чтобы использовать CasperJS, сначала нужно установить последнюю версию PhantomJS, которую можно получить с сайта **https://code.google.com/p/phantomjs/downloads/list**. Проще всего загрузить уже откомпилированную бинарную версию для вашей операционной системы. Самостоятельная компиляция из исходного кода довольно трудна и нужна в редких случаях (например, если у вас старая версия Linux или же вам нужно отключить SSE-оптимизацию и т.д.).

Получить последнюю версию CasperJS (1.1 на данный момент) можно с сайта **http://casperjs.org/**.

Далее приводится CasperJS-версия приведенного ранее теста для поисковика Yahoo!:

```
var casper = require('casper').create();
casper.start('http://search.yahoo.com/', function() {
        this.fill('form#sf', { "p": 'JavaScript' }, false);
        this.click('#yschbt');
});
casper.then(function() {
        this.test.assertExists('#resultCount', 'Got result count');
});
casper.run(function() {
        this.exit();
});
```

Теперь давайте разберемся, как запустит этот сценарий CasperJS:

```
% bin/casperjs yahooSearch.js
PASS Got result count
%
```

Как видите, все очень просто. И это значительно быстрее соединения с удаленным сервером Selenium, с его порождением и уничтожением браузера. Все это работает в реальном WebKit-браузере, но учтите, что версия WebKit, используемая Apple в Safari, и версия WebKit, которая используется Google в Chrome, отличаются от той, которая запускается здесь в PhantomJS.

А что насчет скриншота?

```
var casper = require('casper').create();
casper.start('http://search.yahoo.com/', function() {
        this.fill('form#sf', { "p": 'JavaScript' }, false);
        this.click('#yschbt');
});
casper.then(function() {
  this.capture('results.png', {
        top: 0,
        left: 0,
        width: 1024,
        height: 768
  });
  this.test.assertExists('#resultCount', 'Got result count');
});
casper.run(function() {
        this.exit();
});
```

Рис. 6.2. Создание снимка экрана с помощью CasperJS

Код-«захватчик экрана» может захватить не только весь экран, но и только заданный CSS-селектор вместо захвата всей страницы.

Этот снимок экрана подобен тому, который мы получили с использованием Selenium. Наибольшее отличие заключается в размере снимка экрана: CasperJS не «снимает» всю веб-страницу, как это делает Selenium, а только «видимую» ее часть.

У CasperJS есть много разных полезных приемов. Например, давайте рассмотрим автоматический экспорт результатов тестов в файл, отформатированный как JUnit XML. Полный сценарий наш будет таким:

```
var casper = require('casper').create();
casper.start('http://search.yahoo.com/', function() {
  this.fill('form#sf', { "p": 'JavaScript' }, false);
  this.click('#yschbt');
});
casper.then(function() {
  this.capture('results.png', {
    top: 0,
    left: 0,
    width: 1024,
    height: 768
  });
  this.test.assertExists('#resultCount', 'Got result count');
});
casper.run(function() {
  this.test.renderResults(true, 0, 'test-results.xml');
});
```

Файл `test-results.xml` теперь будет содержать вывод теста в формате JUnit XML. Как уже неоднократно упоминалось, этот формат хорош тем, что его понимают множество утилит сборки, включая Hudson/Jenkins. Далее приводится содержимое файла `test-results.xml` после запуска данного теста:

```
<testsuite>
        <testcase classname="ss" name="Got result count">
        </testcase>
</testsuite>
```

Вывод на консоль будет таким:

```
PASS 1 tests executed, 1 passed, 0 failed.
Result log stored in test-results.xml
```

6.4. Тестирование производительности

Основное внимание при тестировании производительности сосредотачивается на тестировании того, как быстро ваше веб-приложение загружается. Формат HAR (HTTP Archive) является стандартным для захвата этой информации; а его спецификация доступна в блоге Яна Одварко (**http://www.softwareishard.com/blog/har-12-spec/**).

По своей сути HAR — это JSON-отформатированный объект, который может быть просмотрен и проанализирован многими инструментами, включая бесплатные онлайн средства просмотра. В рамках мониторинга производительности вашего веб-приложения вам нужно генерировать HAR-файлы, в которых фиксируются данные о HTTP-взаимодействии приложения, а затем как-то анализировать эти данные на наличие проблем. Захват и мониторинг данных в течение долгого времени позволяет достичь понимания производительности вашего приложения.

Создание HAR-файлов

Для создания HAR-файлов используется два основных подхода, под которые доступно несколько утилит. Самый простой и гибкий подход состоит в использовании программируемого прокси, который перехватывает HTTP-запросы и ответы. Прокси может работать с любым браузером, в том числе с браузерами мобильных устройств, что и предоставляет вам максимальную гибкость.

Другой подход состоит в использовании сниффера пакетов вроде **tcpdump** для захвата HTTP-трафика в формате PCAP, а затем использовании утилиты вроде **pcap2har** для создания HAR из PCAP-файлов. Такое решение, возможно, даст "более правильные" показатели производительности, поскольку между сервером и клиентом не будет ничего лишнего в виде прокси. Кроме того, несмотря на то, что метод со сниффером более сложный, в некоторых случаях вам просто придется его использовать, поскольку применить метод с прокси будет невозможно. Например, мобильные браузеры не всегда позволяют устанавливать прокси, поэтому для создания

HAR мобильных устройств требуется пакетный сниффер (подробнее об этом можно почитать на странице https://code.google.com/p/pcaphar/wiki/CaptureMobileTraffics).

Использование прокси

Чтобы понять, как работает прокси, мы будем использовать программируемый прокси с открытым исходным кодом browsermob (https://github.com/webmetrics/browsermob-proxy). Написаный на Java, этот прокси легко интегрируется в сценарии Selenium WebDriver и обладает некоторыми превосходными программируемыми функциями, а также интерфейсом REST[2]. Далее мы разберемся, как использовать **browsermob** для захвата HTTP и создания HAR-файла.

Примечание.

Не нужно рассматривать browsermob только как средство получения HAR. Это полнофункциональный прокси, среди функций которого есть и черные/белые списки сайтов, ограничение скорости загрузки и даже установка пользовательского DNS.

По умолчанию **browsermob** запускается на порту 8080. Это не только порт прокси, это также порт взаимодействия REST-интерфейса, поэтому не изменяйте его, если вы не понимаете, что делаете.

После загрузки `browsermob-proxy` вы можете запустить его так:

```
% /bin/sh bin/browsermob-proxy
```

Теперь вы можете взаимодействовать с ним, запускать, останавливать и т.д. Я установил привязку JavaScript для browsermob-прокси, которая доступна как npm-пакет:

```
% npm install browsermob-proxy
```

Теперь мы готовы к созданию HAR-файлов на JavaScript! Для начала рекомендуется использовать браузеры Firefox и Internet Explorer, поскольку это единственные браузеры, настройки прокси которых Selenium WebDriver может изменить программно. У вас должен быть запущен Selenium Server для запуска Firefox или Internet Explorer (автономно или в сетке).

2 К сожалению, на момент написания этих строк далеко не все функции доступны через интерфейс REST.

NPM-модуль `browsermob-proxy` может генерировать HAR двумя способами:

- Простой способ.
- Расширенный способ.

Давайте начнем с простого способа, который предпочтителен, если у вас есть один URL, водопадный график которого нужно просмотреть. Следующий код генерирует HAR-файл для Yahoo.com:

```
var Proxy = require('browsermob-proxy').Proxy
        , fs = require('fs')
        , proxy = new Proxy()
;
proxy.doHAR('http://yahoo.com', function(err, data) {
        if (err) {
                console.error('ERROR: ' + err);
        } else {
                fs.writeFileSync('yahoo.com.har', data, 'utf8');
        }
});
```

В приведенном коде мы загружаем модуль `browsermob-proxy` и создаем новый объект `Proxy`. Поскольку у этого объекта нет параметров, считается, что прокси `browsermob-proxy` запущен на *localhost* (порт 8080). Далее мы просто передаем URL и обратный вызов. Если нет ошибок, второй параметр нашего обратного вызова и будет HAR-данными для этого сайта. Нам остается только записать эти данные в файл.

Если ваш прокси не работает на *localhost* или вы используете другой порт, просто укажите это в конструкторе:

```
, proxy = new Proxy( { host: 'some.other.host', port: 9999 } )
```

Помните, что вам нужна версия 2.x сервера Selenium. При этом по умолчанию подразумевается, что Selenium Server работает на *localhost* и использует порт 4444. Если вам требуется задать другой порт, это можно сделать в конструкторе Proxy:

```
, proxy = new Proxy( { selHost: 'some.other.host', selPort: 6666 } )
```

Теперь давайте рассмотрим расширенный способ создания HAR-файлов. Данный подход нужно использовать, если требуется создать HAR-файл для более сложного веб-взаимодействия, чем просто загрузка страницы. Используя этот метод, вы можете установить прокси, он начнет захват данных, а модуль `browsermob-proxy` будет связующим звеном между кодом Selenium и прокси.

Для этого примера мы будем использовать ранее приведенный пример использования Selenium **webdriverjs** загрузки страницы Yahoo! и поиска строки "JavaScript". Сначала мы создадим HAR для всего этого взаимодействия:

```
var Proxy = require('browsermob-proxy').Proxy
        , webdriverjs = require("webdriverjs")
        , fs = require('fs')
        , proxy = new Proxy()
;
/*
* Прокси вызывается с "именем" для этого сеанса, функцией Selenium
* для захвата и обратным вызовом, который запишет HAR-данные
* в файл или выведет на консоль сообщение об ошибке
*/
proxy.cbHAR('search.yahoo.com', doSeleniumStuff, function(err, data) {
        if (err) {
                console.error('Ошибка захвата HAR: ' + err);
        } else {
                fs.writeFileSync('search.yahoo.com.har', data, 'utf8');
        }
});
/*
* Это функция Selenium, которой передается прокси webdriverjs и обратный вызов,
* который должен быть вызван, как только взаимодействие будет завершено.
*/
function doSeleniumStuff(proxy, cb) {
        var browser = webdriverjs.remote({
                host: 'localhost'
                , port: 4444
                , desiredCapabilities: {
                        browserName: 'firefox'
                        , seleniumProtocol: 'WebDriver'
                        , proxy: { httpProxy: proxy }
```

```
                }
});
// Далее все, как обычно. Вы можете запустить обычный
// тестовый код или какой-то специальный, HAR которого
// вам нужно получить
// Просто установите обратный вызов browsermob-proxy в 'end()'
browser
        .testMode()
        .init()
        .url("http://search.yahoo.com")
        .setValue("#yschsp", "JavaScript")
        .submitForm("#sf")
        .tests.visible('#resultCount', true, 'Got result count')
        .saveScreenshot('results.png')
        .end(cb);
}
```

Этот метод подразумевает, что `browsermob-proxy` запущен на том же хосте (*localhost*) и порту (8080), что и сценарий. Как уже было сказано, вы можете изменить данные значения в конструкторе `Proxy`. Этот метод также позволяет вам управлять взаимодействием Selenium, поэтому вы больше не нуждаетесь в параметрах `selHost` и `selPort` конструктора `Proxy`, если автономный сервер (`selenium-server-standalone-2.xx.x.jar`) или сетка не работают на 4444 порте узла *localhost*.

Кроме того, не забывайте, что динамическая инъекция прокси Selenium работает только в Firefox и в Internet Explorer с использованием WebDriver (но не в Selenium1/Remote Control), таким образом, получается, что ваш объект `webdriverjs` должен использовать один из этих двух браузеров.

NPM-пакет **browsermob-proxy** также позволяет вам указать пропускную способность и задержку. Это полезно не только для генерации HAR, но и для тестирования медленного соединения, например, вы можете узнать, как будет загружаться ваш сайт с модема 56К и с задержкой 200 мс. Задать параметры пропускной полосы можно с помощью ключей `downloadKbps` и `uploadKbps`, а задержку — с помощью ключа `latency`:

```
, proxy = new Proxy({
        downloadKbps: 56
```

```
        , uploadKbps: 56
        , latency: 200
    })
```

Таким образом вы сможете протестировать низкоскоростное соединение.

Нужно отметить, что **browsermob-proxy** — не единственное прокси-решение. Вы можете использовать и альтернативные инструменты, такие как Fiddler (http://fiddler2.com/home) и Charles (http://www.charlesproxy.com/), но, на мой взгляд, `browsermob-proxy` является оптимальным решением: бесплатным и отлично работающим.

Это мы рассмотрели, как прикрутить прокси к Selenium для генерации HAR-файлов. То же самое можно сделать и для PhantomJS и CasperJS, они также могут генерировать HAR-файлы. Все что вам нужно, это также установить прокси.

Модуль **browsermob-proxy** изначально написан для Node.js, а PhantomJS/CasperJS очень отличаются от Node.js, хоть на первый взгляд все они выглядят похоже. Поэтому сначала нужно установить прокси, используя Node.js, затем использовать CasperJS для создания HAR.

В целом, данный процесс подобен расширенному методу, описанному ранее, за исключением того, что передаваемый обратный вызов (callback) для Selenium-части будет порожден сценарием CasperJs для осуществления управления. Как только он будет закончен, мы сможем получить HAR. Когда-нибудь, возможно, PhantomJS и Node.js будут тесно взаимодействовать или же PhantomJS получит код запроса HTTP. А пока мы нуждаемся в Node.js.

Вот как это выглядит со стороны Node.js:

```
var Proxy = require('browsermob-proxy').Proxy
        , spawn = require('child_process').spawn
        , fs = require('fs')
;
var proxy = new Proxy();
proxy.selHAR('MyCoolHARFile', doCasperJSStuff, function(err, data) {
        if (err) {
                console.error('ERR: ' + err);
```

```
        } else {
                fs.writeFileSync('casper.har', data, 'utf8');
        }
});
function doCasperJSStuff(proxy, cb) {
        casperjs = spawn('bin/casperjs'
                , [ '--proxy=' + proxy, process.argv[2] ]);
        casperjs.on('exit', cb);
}
```

Мы используем `proxy.selHAR`, как и в предыдущем примере, но вместо Selenuim-функции мы используем функцию CasperJS. При этом прокси установлен в browsermob. Как только процесс CasperJS завершает работу, мы получаем HAR-файл. Использовать же этот сценарий нужно так:

```
% node casperHAR.js casperScript.js
```

Где `casperHAR.js` — это файл, содержащий предыдущий код, а `casperScript.js` — любой сценарий CasperJS (например, приведенный ранее сценарий для поиска в Yahoo!). Как и в случае с Selenium, вы можете указать имя компьютера и порт, где запущен прокси browsermob, а также ограничить пропускную способность.

Просмотр HAR-файлов

Визуализация данных в HAR-файлах очень проста. Самый легкий способ — использовать онлайн средство просмотра: http://www.softwareishard.com/har/viewer/. Просто перетяните HAR-файл на эту страницу (для этого нужно использовать современный браузер, поддерживающий HTML5), и вы увидите водопадный график HAR-данных. Только перед загрузкой вашего HAR-файла убедитесь, что вы выключили переключатель Validate data before processing?[3].

Рис. 6.3 показывает HAR-файл для нашего примера search.yahoo.com.

Из графика видно, что мы сделали 10 запросов, а полная загрузка страницы заняла целых 8.25 с. Если подвести мышь к каждому запросу, вы увидите, сколько времени занял этот конкретный запрос в следующем порядке: Разрешение DNS (DNS Lookup), Подключе-

3 Я еще ни разу не сталкивался с корректной валидацией HAR-файлов.

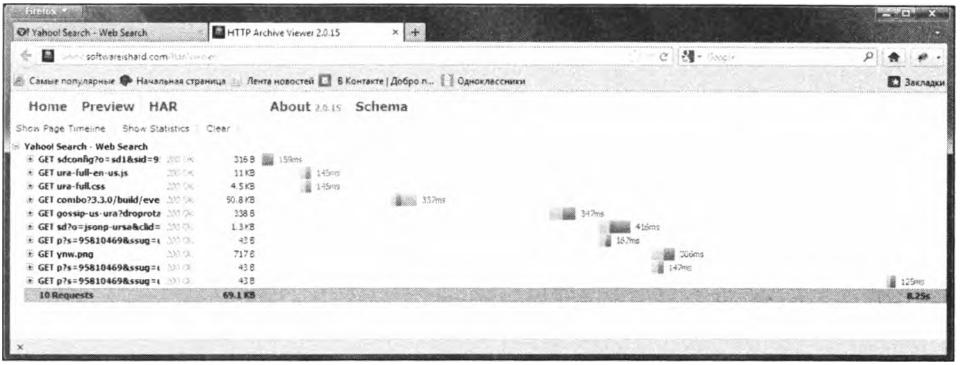


Рис. 6.3. Водопадный график для примера search.yahoo.com

ние (Connecting), Отправка (Sending), Ожидание (Waiting), Получение (Receiving).

Более интересный пример — с Yahoo.com. В примере, показанном на рис. 6.4, для загрузки страницы было сделано гораздо больше запросов — 81, а полная загрузка этой страницы заняла целых 12.3 секунд. Обратите внимание и на размер страницы. В предыдущем примере размер был чуть больше 69 Кб, а вот полная страница Yahoo.com заняла более 1 Мб — на ней много картинок, баннеров и т.д.

Это мы рассмотрели онлайн-сервис просмотра HAR-файлов, но есть средства, позволяющие обойтись без загрузки их на удаленный веб-сайт. Одно из таких средств — программа с открытым исходным кодом HAR Viewer, которую можно загрузить по следующему адресу: **https://code.google.com/p/harviewer/downloads/list**.

Еще одно средство просмотра HAR-файлов — **YSlow** (http://developer.yahoo.com/yslow/) – доступно как в виде плагина браузера, так и в виде утилиты командной строки, которая принимает HAR-файлы в качестве входных данных и выдает на выходе XML, простой текст, JSON (**http://yslow.org/command-line-har/**). Запустить YSlow из командной строки или непосредственно из сценария Node.js можно с помощью следующей команды:

```
% npm install yslow -g
```

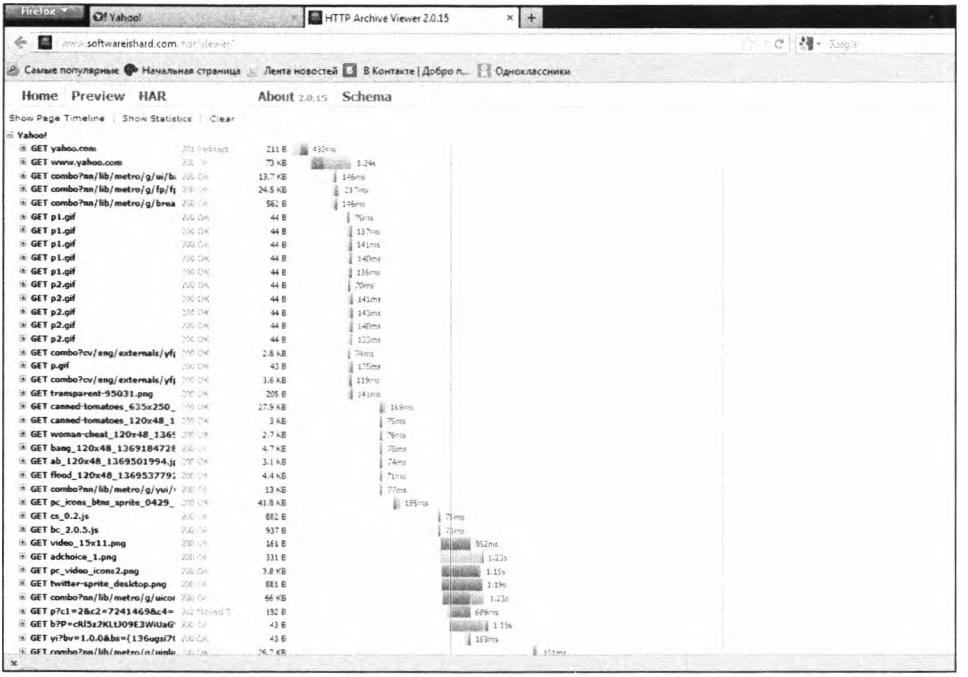


Рис. 6.4. Водопадный график для примера Yahoo.com

Чтобы просмотреть графическое представление вывода YSlow, вам нужна другая программа — **YSlow visualizer** (визуализатор YSlow), скачать которую можно по адресу: **http://www.showslow.com/beacon/yslow/**. Данную программу можно также установить локально (**http://www.showslow.org/Installation_and_configuration**), если вы не хотите отправить ваши YSlow-данные третьей стороне.

Давайте посмотрим, как это все работает вместе. После установки YSlow передайте ему HAR-файл для просмотра результатов на Showslow.com:

```
% yslow -i all -b http://www.showslow.com/beacon/yslow/ yahoo.com.har
```

Данная команда отправит ваш HAR-вывод из Yahoo.com на ShowSlow.com для визуализации. Затем нам нужно перейти на **http://www.showslow.com/** и найти ваш URL в списке последних адресов, чтобы просмотреть полный графический вывод (**http://www.showslow.com/details/6005/http://www.yahoo.com/**).

Множество других пользователей также создавали собственные YSlow-графики Yahoo.com, поэтому вы можете просмотреть все созданные ранее (другими пользователями) графики. В нижней части страницы вы также обнаружите ссылку на HAR Viewer для Yahoo.com — замкнутый круг!

Вы можете генерировать визуализации самостоятельно, используя опцию `-beacon` (см. **http://yslow.org/user-guide/**). При желании можно даже запустить собственный локальный сервер Showslow.

Файлы HAR очень интересны, но, конечно, они не дают полной картины производительности. Поэтому далее мы поговорим о тестировании производительности браузера.

Тестирование производительности браузера

HAR-файлы – это, конечно, хорошо, и создавать их просто, но они не дают полной картины производительности. Ведь помимо сетевой активности браузеры должны также парсить HTML-код, парсить[4] и выполнить JavaScript-код, сформировать и расположить элементы в окне браузера и т.д., а все это занимает время. Сколько времени? Существуют инструменты, позволяющие точно измерить все то, что делает браузер во время работы вашего приложения. Мы в рамках данной книги отметим два из них:

1. **dynaTrace AJAX Edition (http://www.compuware.com/application-performance-management/ajax-performance-testing.html)** для Firefox и Internet Explorer.
2. **Speed Tracer (https://developers.google.com/web-toolkit/speedtracer/)** для Chrome.

Оба инструмента выполняют одни и те же задачи: во время загрузки страницы они собирают информацию, которая сразу может быть отображена в виде графика или же сохранена для просмотра в дальнейшем. При просмотре графического вывода вы можете идентифицировать проблемы производительности и затем исправлять их. Процесс сбора низкоуровневых таймингов специфичен для каждого браузера, следовательно, не может быть универсального коллек-

4 Англ. parse – 1) синтаксический анализ, синтаксический разбор; грамматический разбор; 2) анализировать, разбирать.

тора вроде HAR-прокси, и именно потому у нас два инструмента под разные браузеры.

В данном разделе мы вкратце рассмотрим Speed Tracer, представляющий собой расширение для браузера Google Chrome и позволяющий увидеть, на что тратится время в работающем приложении.

Установите расширение и затем нажмите небольшую зеленую кнопку в верхнем правом углу экрана, чтобы получить доступ к пользовательскому интерфейсу Speed Tracer. Сразу можно заметить красную кнопку, которая запускает запись данных о сетевом взаимодействии, парсинге HTML и JavaScript, а также о прорисовке в окне браузера. Вся эта информация представляется в знакомом нам водопадном графике.

Сверху интерфейса Speed Tracer находится график медлительности (Sluggishness) и график сети (Network). Первый график более интересен и визуализирует, насколько быстро реагирует (или не реагирует) пользовательский интерфейс вашего приложения. Высокие пики на этом графике говорят о том, что поток пользовательского интерфейса (UI thread) браузера был блокирован — и это очень плохо!

Наконец, Speed Tracer предоставляет подсказки относительно проблемных областей в вашем приложении, например, Speed Tracer отметит событие, длящееся дольше, чем 100 мс. Это обусловлено тем, что обработка событий блокирует основной поток пользовательского интерфейса (UI thread), поэтому события, длящиеся дольше, чем 100 мс, сделают интерфейс «тормозным».

Еще одна важная кнопка – кнопка сохранения, позволяющая сохранить данные и исследовать их позже, когда у вас будет время. На самом деле это очень важная функция, которая может быть очень полезной в процессе автоматической сборки. Как часть нашей сборки, мы можем автоматически генерировать данные Speed Tracer для нашего приложения, чтобы убедиться, что никакие изменения не влияют на производительность (см. гл. 8). Рис. 6.5 показывает процесс анализа yahoo.com с помощью Speed Tracer.

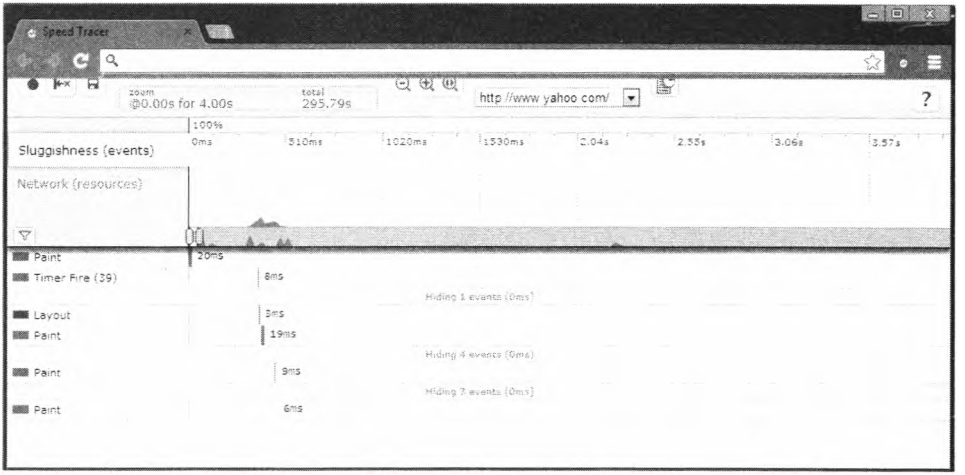


Рис. 6.5. Speed Tracer UI анализирует Yahoo.com

6.5. Нагрузочное тестирование

Тестирование производительности призвано выявить узкие места в вашем коде, негативно влияющие на производительность приложения, и ответить на вопрос «Как быстро ваше приложение может работать?». Нагрузочное тестирование пытается определить, сколько данных может обработать ваше приложение. В значительной мере это подобно тестированию производительности, и фактически можно утверждать, что тестирование производительности — это подмножество нагрузочного тестирования. Тестирование производительности определяет, что делает ваше приложение при самых незначительных нагрузках, а нагрузочное тестирование определяет, как выполняется ваше приложение под максимальной нагрузкой.

То есть если вы захотите понять, что будет с вашим сайтом/веб-приложением, если на него одновременно зайдут миллион пользователей, то тут вам и понадобится нагрузочное тестирование. При этом в качестве определяемых критических значений далеко не обязательно будет число запросов в секунду до полного отказа. Считается, что правильнее определять максимально выдерживаемую нагрузку (число запросов, которое приложение может обра-

ботать) не до отказа, а до того момента, пока не будет превышено некоторое, заданное вами, время отклика.

Таким образом, задачей нагрузочного тестирования является не только убедиться в способности обрабатывать большие нагрузки в рамках указанного времени отклика, но и определить максимально приемлемое для вас время отклика. Вся эта информация потом может быть использована для определения стратегии масштабирования (горизонтальное масштабирование, вертикальное масштабирование) в случае необходимости повысить производительность перед лицом больших нагрузок. Например, может, будет вполне достаточно увеличить мощность процессора на сервере (вертикальное масштабирование), вместо того чтобы переписывать код, распараллеливая ваше приложение на несколько серверов (горизонтальное масштабирование).

Нагрузочное тестирование применяется не обязательно для всего приложения. Вы можете протестировать отдельные его функции для определения их производительности.

При нагрузочном тестировании очень важно учитывать загрузку памяти, диска и процессора. Возможно, ограниченные значения какого-либо из этих параметров в конечном счете и послужат причиной критичного увеличения времени отклика при максимальной нагрузке.

Нагрузочное тестирование браузера

Нагрузочное тестирование веб-приложения в браузере обычно представляет собой отправку вашему приложению большого количества трафика (как можно больше) и контроль изменений в его производительности. Измерение производительности обычно означает контроль времени отклика: вы запрашиваете страницу методом `GET` или отправляете форму методом `POST` и отслеживаете, сколько времени требуется приложению на ответ.

Лучшая утилита для этого типа тестирования — Apache Bench (**http://httpd.apache.org/docs/2.4/programs/ab.html**). Утилита командной строки `ab` принимает множество опций для запросов `GET`, `POST`, `PUT` и `DELETE`, а результаты помещает в CSV-файл. В

этой главе мы будем работать с ее Node.js-версией, которая называется nodeload (**https://github.com/benschmaus/nodeload**).

Первым делом установим утилиту:

```
% sudo npm install nodeload -g
```

Примечание

Таким образом, вы не установите nodeload ни в Windows, ни в Linux. Через npm install программа почему-то отказывается устанавливаться, что в Windows, что в Linux. Рассмотрим, как правильно установить nodeload в Linux. Первым делом нужно установить сам npm:

```
sudo apt-get install npm
```

Затем нам нужно установить программу git, необходимую для загрузки исходного кода nodeload:

```
sudo apt-get install git
```

После чего загружаем саму программу nodeload:

```
git clone git://github.com/benschmaus/nodeload.git
cd nodeload
```

В каталоге nodeload, в который мы только что перешли, будет исполнимый файл nl.js. Откройте его. Первая строчка этого файла будет такой:

```
#!/usr/bin/env node
```

В Linux исполнимый файл node называется nodejs, поэтому эту строчку нужно изменить так:

```
#!/usr/bin/env nodejs
```

После этого можно приступить к использованию программы, как описано далее в книге.

Рассмотрим формат использования утилиты `nl.js` (с утилитой `ab` (исполнимый файл Apache Bench) все примерно то же самое):

```
% nl.js
1 Aug 20:28:52 - nodeload.js [options] <host>:<port>[<path>]
Available options:
-n, --number NUMBER        Number of requests to make.
Defaults to value of --concurrency unless a time limit is
specified.
-c, --concurrency NUMBER   Concurrent number of connections.
Defaults to 1.
-t, --time-limit NUMBER    Number of seconds to spend running
test. No timelimit by default.
```

```
-e, --request-rate NUMBER  Target number of requests per
seconds. Infinite by default
-m, --method STRING        HTTP method to use.
-d, --data STRING          Data to send along with PUT or
POST request.
-r, --request-generator STRING Path to module that exports
getRequest function
-i, --report-interval NUMBER Frequency in seconds to report
statistics. Default is 10.
-q, --quiet                Supress display of progress
count info.
-h, --help                 Show usage info
```

Типичное использование этой утилиты выглядит примерно так:

```
% nl.js -c 10 -n 1000 http://yhoo.it/XUdX3p
```

Данная команда запросит Yahoo.com 1000 раз по 10 запросов за один раз. Вот полученные мною результаты:

```
% nl.js -c 10 -n 1000 http://yahoo.com
http.createClient is deprecated. Use `http.request` instead.
Completed 910 requests
Completed 1000 requests
Server:                                yahoo.com:80
HTTP Method:                           GET
Document Path:                         /
Concurrency Level:                     10
Number of requests:                    1000
Body bytes transferred:                207549
Elapsed time (s):                      11.19
Requests per second:                   89.37
Mean time per request (ms):            111.23
Time per request standard deviation:147.69
Percentages of requests served within a certain time
(ms)
  Min: 86
  Avg: 111.2
  50%: 94
  95%: 113
  99%: 1516
  Max: 1638
```

Из отчета видно, что все эти 1000 запросов заняли примерно 11.2 мс, но один из них занял 1.638 с (см. `Max`). Было бы хорошо включить сюда серверную статистику по использованию процессора, памяти и диска (накопителя). Для этого, в частности, желательно увеличить продолжительность тестирования, подав на сервер более длительную нагрузку. Тем самым мы «убьем двух зайцев»: и по нагрузке посмотрим, как ведет себя сервер, и за параметрами использования устройств сервера понаблюдаем. Пользоваться будем все тем же nodeload, который вполне может записывать значения использования процессора, памяти и диска. По ходу выполнения теста мы сможем наблюдать «живой» график производительности веб-сервера — график будет изменяться по ходу работы сервера.

Итак, приступим. Первым делом нужно отправить серверу много запросов. Мы отправим 100 000 запросов:

```
% nl.js -c 10 -n 100000 -i 2 http://localhost:8080
```

Данный пример предполагает, что у вас развернут веб-сервер на порту 8080[5]. Сразу после этой команды, не дожидаясь завершения тестирования, откройте браузер и введите адрес: http://localhost:8000. Программа nodeload на время своей работы запускает веб-сервер на http://localhost:8000, отображающий график производительности указанного веб-сервера (в нашем случае — yahoo.com), см. рис. 6.6.

По окончании выполнения тестирования результаты будут сохранены в текущем каталоге в форматах HTML и JSON.

Примечание.

Вы можете контролировать то, что происходит на стороне сервера, используя ваш любимый генератор статистики (`vmstat`, `iostat`, `uptime`, `ps` и т.д.), а nodeload создаст красивый график и сохранит результаты. Еще проще добыть серверную статистику можно в Linux, используя файловую систему /proc.

Важно отметить, что утилита **nodeload** позволяет применять конструкцию зацикливания Loop, которая будет выполнять некоторый заданный код с заданной частотой. Например, в следующей конструкции мы запускаем функции `getMemory` и `getCPU` (получающие

5 Если же веб-сервера у вас нет, можно обратиться к yahoo.com:
% nl.js -c 10 -n 100000 -i 2 http://yahoo.com.

Рис. 6.6. График производительности сервера, созданный nodeload

информацию об использовании памяти и процессора) каждые пять секунд:

```
var myLoop = new loop.Loop(
        function(finished, args) {
                getMemory();
                getCPU();
                finished();
        }
        , [] // Без аргументов
        , [] // Без условий
        , .2 // Каждые 5 секунд
);
myLoop.start();
```

Последний параметр в предыдущем коде — это максимальное число итераций цикла в секунду.

Заголовок следующего сценария устанавливает все наши переменные:

```
var reporting = require('nodeload/lib/reporting')
        , report = reporting.REPORT_MANAGER.getReport('System Usage')
        , memChart = report.getChart('Memory Usage')
        , cpuChart = report.getChart('CPU Usage')
        , loop = require('nodeload/lib/loop')
        , fs = require('fs')
;
```

Можно видеть, что здесь я определил один отчет (`'System Usage'`) с двумя графиками: `'Memory Usage'` (использование памяти) и `'CPU Usage'` (использование процессора). Функция `getMemory` использует файловую систему Linux /proc для получения информации об использовании памяти:

```
function getMemory() {
        var memData = getProc('meminfo', /\s*kb/i);
        memChart.put({ 'Free Memory': memData['MemFree']
                , 'Free Swap': memData['SwapFree'] });
        report.summary['Total Memory'] = memData['MemTotal'];
        report.summary['Total Swap'] = memData['SwapTotal'];
}
```

Конечно, мы можем извлечь любые значения, которые нам нужны. Функция `getProc()` не показана, она просто берет файл в файловой системе /proc и объектизирует его.

Функция `getCPU` в целом аналогична функции `getMemory`, приведенной выше:

```
function getCPU() {
        var meminfo = fs.readFileSync('/proc/loadavg', 'utf8')
                , vals = meminfo.split(/\s+/)
                , cpuInfo = getProc('cpuinfo')
        ;
        cpuChart.put( {
                '1 Min': vals[0]
                , '5 Min': vals[1]
                , '15 Min': vals[2]
        });
        report.summary['CPU'] = cpuInfo['model name'];
}
```

Снова, мы захватываем некоторые значения и добавляем их в график CPU. Запустите сценарий и перейдите на порт 8000 (протокол HTTP), чтобы просмотреть «живые» результаты. Сводные данные выводятся справа, а обновляющиеся «живые» графики находятся в основной колонке. По завершении работы сценария в локальном каталоге будет создан HTML-файл с полным окончательным графиком.

В завершение данного раздела отметим, что у **nodeload** есть много других "фишек" и полезных возможностей, в том числе множественные циклы для параллельного выполнения циклов, пакет статистики для дальнейшего анализа наборов данных и очень гибкий, программируемый способ указывать запросы. С помощью nodeload вы можете отобразить в виде графика любое тестирование.

6.6. Отслеживание использования ресурсов

Отслеживание использования памяти и центрального процессора вашим приложением может быть особенно интересно в двух случаях: одностраничное JavaScript-приложение на стороне клиента и демон (служба) на сервере. Если пользователь не остается на одной странице в течение долгого времени, использование памяти JavaScript, вероятно, не имеет значения (конечно, есть исключения). А вот высокое использование процессора говорит о том, что в браузере происходит большая обработка. Современные браузеры, такие как Chrome, прекрасно изолируют память и ресурсы процессора разных страниц друг от друга. Однако, одностраничное веб-приложение должно помнить и об использовании процессора, и об использовании памяти. Ведь нужно заботиться не только о ресурсах клиента, но и о ресурсах сервера. К тому же некоторые настройки сервера могут прервать выполнение вашего веб-приложения, если будут превышены определенные лимиты.

Например, по умолчанию на многих веб-серверах установлено максимальное время выполнения сценария в 30 секунд, а лимит памяти — 64-128 Мб. Если ваше приложение будет выполняться дольше, чем 30 секунд или же оно превысит установленный лимит памяти, его выполнение будет прекращено.

При использовании современных браузеров основной канал утечки памяти в JavaScript заключается в хранении ссылок на объекты, в которых вы больше не нуждаетесь. Также утечка памяти может произойти при выделении памяти для больших объектов, которые ни разу не использовались.

Рассмотрим пример:

```
function createClosure() {
  var bigHairyVariable = { ... }, wow = 'wow';
  return {
    a: bigHairyVariable
    , b: wow
    , c: 87
    , d: function() { return bigHairyVariable; }
  };
}
// никогда не используется global.a — потраченная впустую память — утечка?
var global = createClosure();
```

Конечно, здесь утечка памяти не так явно видна, как в языках вроде C, где память явно выделяется. Фактически это может вообще не быть утечкой, если использование переменной `bigHairyVariable` запланировано в будущем.

Переменные объекта и функции в JavaScript являются ссылками. В предыдущем примере `bigHairyVariable` выходит за пределы области действия, когда функция возвращается, но на ее память все еще ссылаются две другие переменные — `global.a` и `global.d`. Установка `global.a = null` освобождает ссылку на `bigHairyVariable`, но все еще не освобождает память, поскольку на нее все еще ссылается `global.d`. Чтобы освободить память, нужно еще установить `global.d` в `null`:

```
global.d = null;    // или global = null
```

Память освобождается не сразу, а при следующем проходе сборщика "мусора", который освободит все участки памяти, отмеченные для освобождения.

Вызов `createClosure` снова выделит совершенно новый блок памяти для `bigHairyVariable`, и эта память будет храниться до тех пор, пока не будут освобождены все ссылки на нее.

Обратите внимание на то, что если бы мы вызывали эту функцию из другой функции, когда возвращаемое значение вышло из области (когда функция возвращена), то вся память, связанная с ним, была бы отмечена как свободная.

6.6.1. Отслеживание задействования ресурсов на стороне клиента

Для отслеживания использования системных ресурсов на стороне клиента/пользователя нам пригодятся утилиты из набора WebKit Developer Tools, предоставляющие для этого отличные возможности.

Использование памяти

Первым делом давайте посмотрим на общее использование памяти. Откройте ваш браузер Chrome и введите следующий URL в строке адреса: chrome://memory-redirect/.Теперь выберите **Настройка > Инструменты > Диспетчер задач**.В результате вы получите картинку как в обычном системном диспетчере задач, но в пределах браузера: вы можете просмотреть использование процессора, ввода/вывода, состояние сети для каждого плагина и вкладки (см. рис. 6.7).

Рис. 6.7. Диспетчер задач Chrome

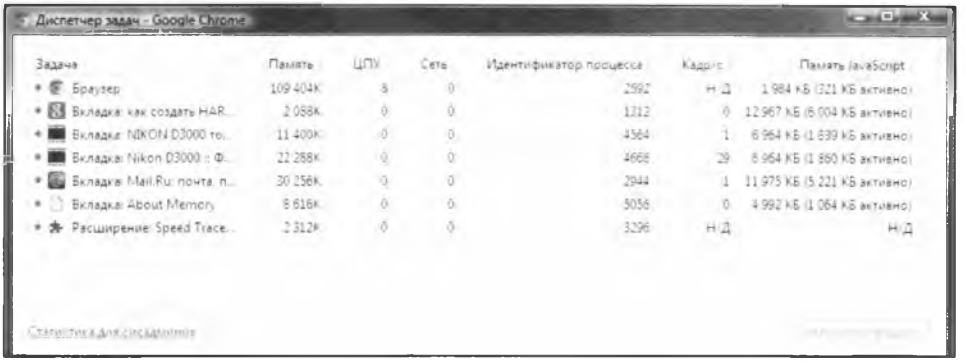


Рис. 6.8. Память JavaScript для каждой вкладки/процесса

При желании вы можете изменить состав отображаемых столбцов, щелкнув правой кнопкой мыши по заголовку таблицы и включив/выключив определенные столбцы. На данный момент для нас самым полезным является столбец **Память JavaScript** (см. рис. 6.8): щелкните правой кнопкой мыши по заголовку таблицы и выберите этот столбец из списка. Посмотрите на статистику, узнайте, на какой странице JavaScript-код потребляет больше всего памяти.

Примечание.

Поскольку все последние версии браузеров поддерживают современные сборщики мусора вместо простого подсчета ссылок, циклы теперь могут быть в плане памяти освобождены должным образом без проблем. Но помимо циклов существует еще множество других потенциальных «пожирателей» памяти. Хороший обзор по данной теме доступен на сайте Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/JavaScript/Memory_Management).

На пути выявления утечек памяти нашей первой остановкой будет график **Memory** на панели **Timeline** (см. рис. 6.9) в окне WebKit Developer Tools.

Рис. 6.9 показывает, что у нас приложение в плане обращения с памятью ведет себя хорошо, — видно, что использование памяти возрастает, а затем снижается, что говорит об освобождении памяти приложением. Если же использование памяти увеличивается и никогда не уменьшается, скорее всего, у вас имеет место утечка памяти.

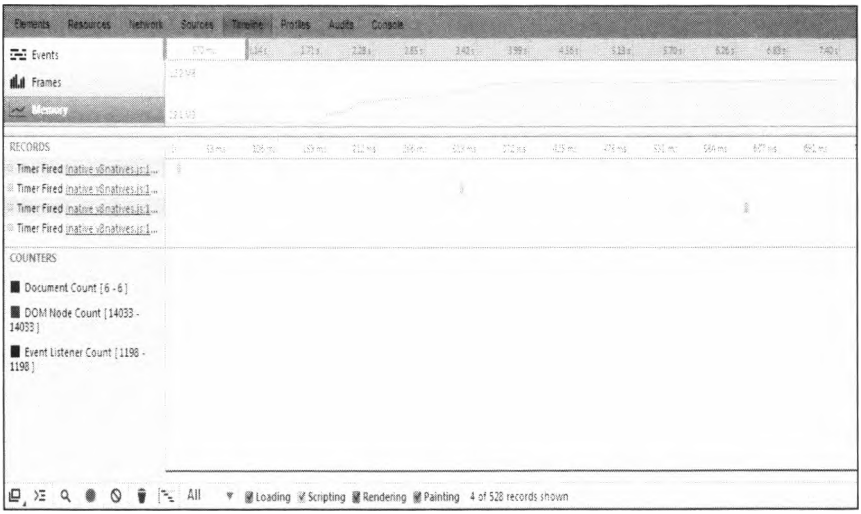


Рис. 6.9. График Memory на панели Timeline

Следующая остановка – вкладка **Profiles** в WebKit Developer Tools (см. рис. 6.10).

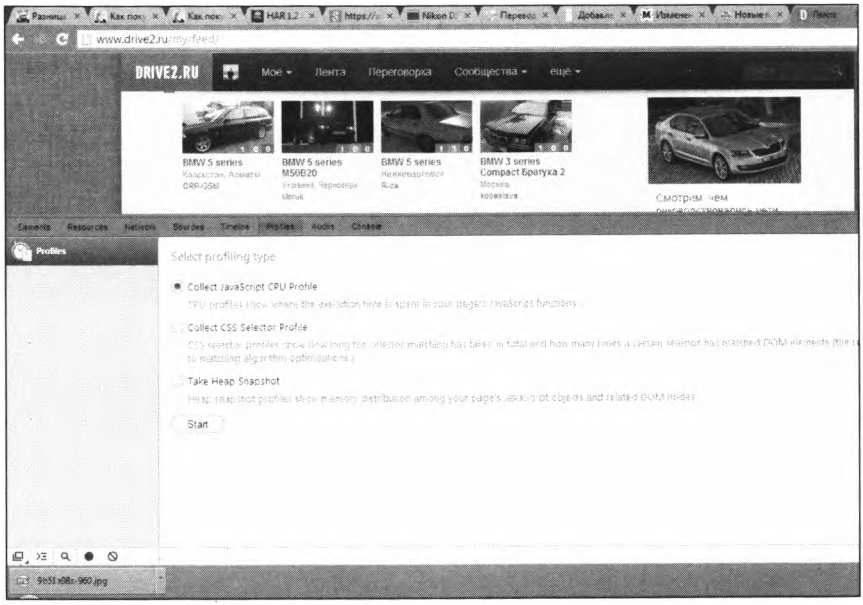


Рис. 6.10. Вкладка Profiles в WebKit Developer Tools

Снимок "кучи" позволяет легко увидеть, сколько памяти занимает ряд объектов JavaScript. При этом WebKit сначала запускает сборщик мусора, чтобы вы получили "чистое" представление фактического использования памяти (см. рис. 6.11). Для удобства можно использовать текстовое поле **Class Filter**, чтобы отфильтровать результирующие объекты.

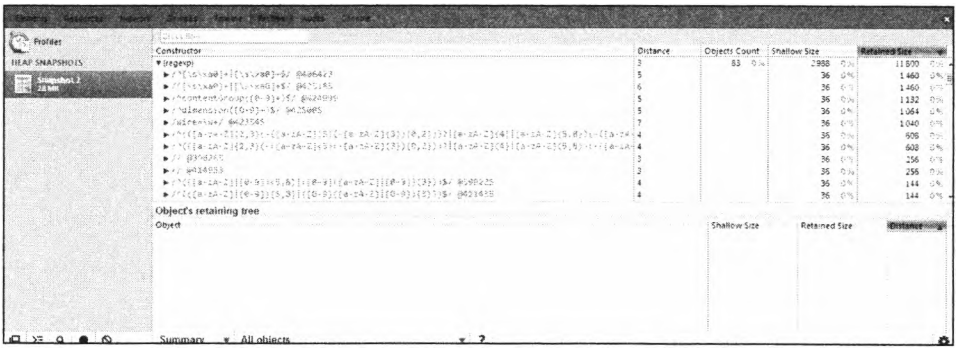


Рис. 6.11. Куча памяти, используемая объектами RegExp

Согласно рис. 6.11, есть более чем 1200 объектов регулярных выражений, «забурившись» дальше, вы сможете просмотреть все эти объекты. Помимо ваших собственных объектов, вы можете также отфильтровать DOM-объекты. Если вы подозреваете, что определенные действия вызывают утечку памяти, сделайте серию снимков для отслеживания динамики. В представлении **Summary** (см. рис. 6.12) вы можете увидеть, какие объекты были размещены между каждым снимком кучи.

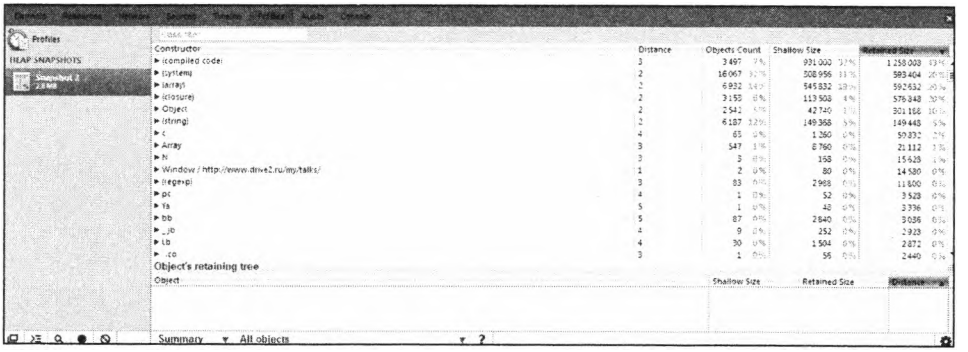


Рис. 6.12. Представление Summary

Переходя от одного снимка к другому в левой части окна, вы можете сравнивать два снимка кучи и находить изменения между ними (см. рис. 6.13).

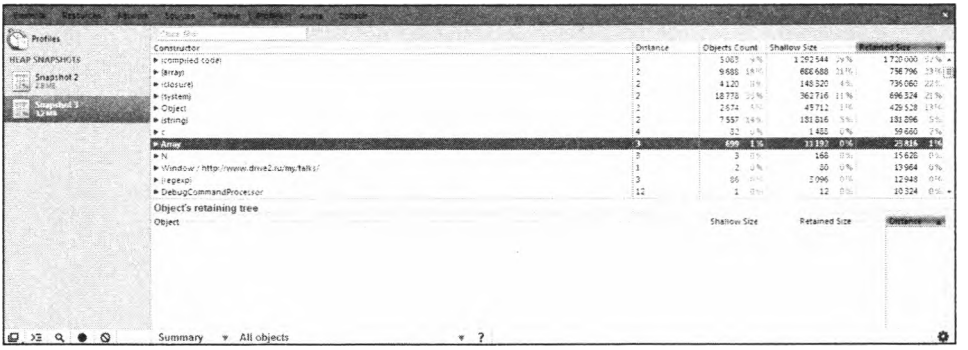


Рис. 6.13. Сравнение между двумя снимками кучи

В приведенном примере есть большая разница в использованной памяти между первым и вторым снимками. Так, например, было создано 699 массивов (**Array**), а до этого массивов было 547. Чтобы определить, какие именно массивы были созданы, вы можете развернуть структуру и просмотреть что к чему.

Примечательно, что в настоящее время нет способа инициировать снимок "кучи" программно. Одно из решений проблемы — вставить оператор отладчика в ваш код и вручную инициировать снимок "кучи", как только интерпретатор остановится.

Примечание.

В качестве доп. информации рекомендую прочитать также о подходе команды Gmail по определению и устранению утечек памяти в их коде по адресу http://bitly.com/PzCbfm. Кроме этого, вы можете использовать средство для обнаружения утечек памяти, созданное самой командой Chrome Developer Tools (https://code.google.com/p/leak-finder-for-javascript/).

Использование процессора

Выбор профиля **JavaScript CPU** в окне WebKit Developer Tools покажет подобную информацию об использовании процессора. Вы

сможете увидеть, сколько времени занимает выполнение каждой функции в вашем веб-приложении и сколько раз вызывается каждая функция. Нажмите **Start** для начала захвата информации об использовании процессора, а затем выполните какие-то действия в вашем веб-приложении. Нажмите **Strop**, и вы получите отчет о задействовании процессора.

Рис. 6.14 показывает, что само приложение заняло 21.3% процессорного времени. Также выводится само процессорное время, которое было использовано приложением. Если щелкнуть на ссылке с названием функции, будет отображен ее фактический код.

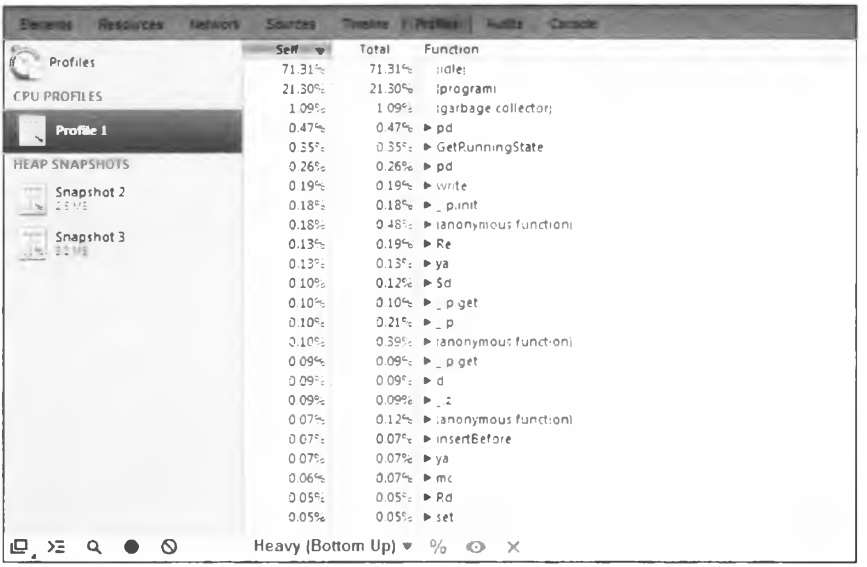


Рис. 6.14. Использование процессора сценарием JavaScript

Нажатие кнопки **%** позволяет переключаться между реальным временем, которое потребовалось для выполнения той или иной функции, и временем, выраженным в процентах (показывает, сколько времени в процентном соотношении заняло выполнение той или иной функции относительно всех остальных функций сценария).

Создавая снимки кучи и собирая информацию об использовании процессора, вы получите полное понимание происходящего на стороне пользователя (клиента) при выполнении вашего приложения.

Глава 7.

Отладка. "Допиливаем" наш фронтенд

Вообще говоря, отладка — это плохо. Я настоятельно не рекомендую тратить на нее время. За исключением тех редких случаев, когда вы вынуждены ею заниматься. Так вот для этих случаев как раз и предназначена данная глава, в которой мы рассмотрим множество отличных утилит и техник, способных помочь вам на ниве «дебагинга».

7.1. Отладка средствами браузера

Отладка JavaScript на стороне клиента неизменно приводит к использованию отладчиков, изначально встроенных в браузеры или добавляемых в них через установку соответствующих расширений. Какой браузер с отладчиком использовать? Наверное, самый распространенный. На сайте StateCounter (**http://gs.statcounter.com**) можно просмотреть текущую долю на рынке каждого браузера во всем мире или в вашем регионе. На рис. 7.1 показан график с января 2016 года до сентября 2019 года.

Конечно, журналы вашего веб-сервера дадут вам самую точную статистику использования браузеров для вашего приложения, но согласно глобальной статистике, показанной на рис. 7.1, Google Chrome является самым популярным браузером. С точки зрения же определенных версий браузера, наиболее популярны версии браузеров IE 11, Chrome 71, Chrome 61, что подтверждает рисунок 7.2, показывающий статистику с января 2016 года до сентября 2019 года. Пунктирная линия на рисунке 7.2 представляет мобильные и планшетные браузеры, и говорит о том, что эти браузеры используются чаще, чем другие.

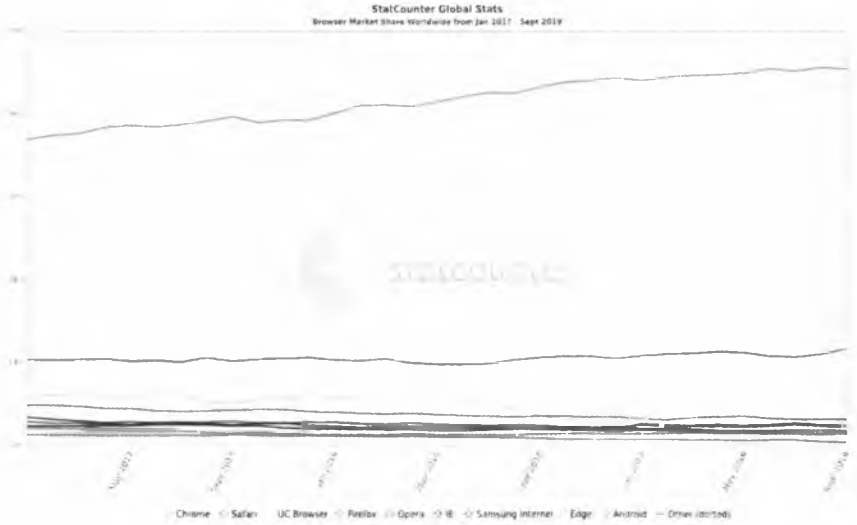


Рис. 7.1. Десять самых популярных браузеров

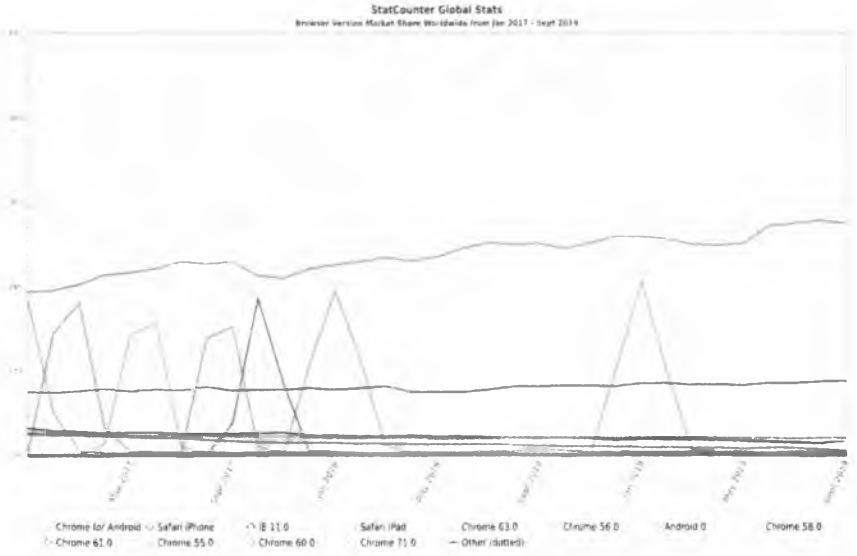


Рис. 7.2. Двенадцать самых популярных версий браузеров

Отладку на мобильных браузерах (в т.ч. и на планшетных браузерах) мы обсудим чуть позже в этой главе. А пока давайте вкратце

взглянем на различные отладчики, предоставляемые каждым настольным браузером. По ходу дела я выделю некоторые различия между ними, но по большей части все они похожи.

Отладчик Firefox

Дедушка всех отладчиков — расширение Firebug браузера Firefox, предоставляющее все необходимые функции для отладки нашего JavaScript-кода. Установите последнюю версию этого расширения (2.0.19 на момент написания этих строк). Перезапустите Firefox и запустите Firebug командой меню **Firefox -> Веб-разработка > Firebug > Открыть Firebug**.
Или просто нажмите F12. На рис. 7.3 показана панель Firebug при просмотре Yahoo.com.

С помощью Firebug можно управлять HTML и CSS, устанавливать точки останова и пошагово выполнять JavaScript. Панель **Сеть** покажет вам знакомый водопадный график загружающихся элементов. Этот график отображает разрывы после загрузок JavaScript.

Примечание.

Кроме панели **Сеть** не забывайте о средствах Speed Tracer (в Chrome) и dynaTrace (в Windows), рассмотренных нами в предыдущей главе.

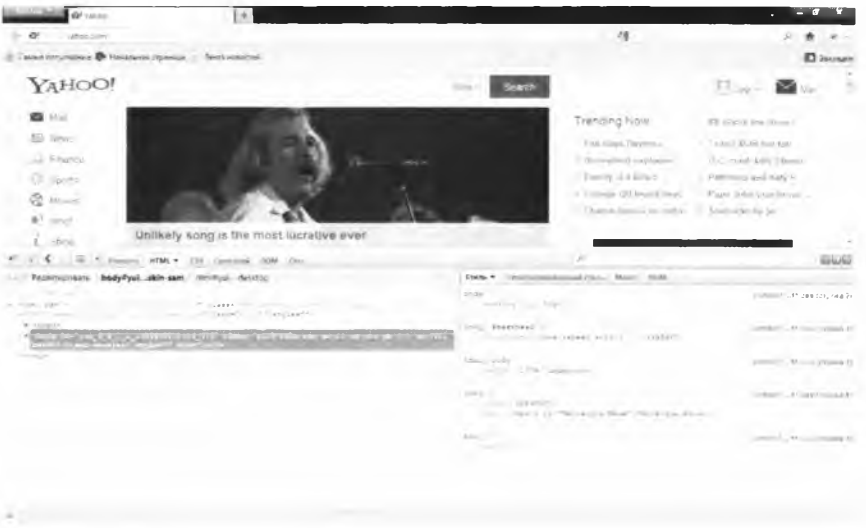


Рис. 7.3. Панель Firebug при просмотре Yahoo.com

Firebug также предоставляет объект `window.console`, который можно использовать для протоколирования сообщений на консоль. Также консоль позволяет точно замерять время, используя методы `console.time(<строка>)` и `console.timeEnd(<строка>)`. Firebug отобразит время, прошедшее между этими двумя вызовами. Получить более подробную информацию о времени можно с помощью методов `console.profile()` и `console.profileEnd()`: Firebug отобразит полную информацию о времени выполнения каждой функции, которая будет вызвана между теми двумя вызовами.

Firebug также предоставляет метод `console.trace()` для трассировки стека, включая все параметры, которые были переданы каждой функции в трассировке. В дополнение к печати строк, вы можете передать любой объект к методам вывода на консоль и Firebug разыменует этот объект для вас, чтобы вывести на экран его поля. Вы можете отобразить основную тайминг-информацию, используя методы console.time(<STRING>) и console.timeEnd(<STRING>) (да, строки должны совпадать, разрешая вам иметь многократные перекрывающиеся синхронизации одновременно). Firebug также позволяет отображать время, которое прошло между двумя вызовами. Вы можете просмотреть более подробные тайминги, используя методы console.profile() и console.profileEnd(). Firebug отобразит полную тайминг-информацию для каждой функции, которая была вызвана между теми двумя вызовами.

Объекты можно анализировать, используя метод `console.dir(<объект>)`, HTML-узлы можно анализировать методом `console.dirxml(<ссылка узла>)`. Наконец, метод `console.assert(<булевое условие>)` позволяет использовать какие-то заданные вами условия и выводить на экран красное сообщение об ошибке, если это условие не будет истинно.

Вы можете выполнить произвольный JavaScript в Firebug или поместить операторы `console*` в ваш код для его отслеживания. Также можно добавить оператор `debugger` для установки точки остановки, но будьте внимательны – не забудьте удалить их по окончании отладки!

Также Firebug позволяет именовать анонимные функции, используя свойство `displayName`. Давайте рассмотрим это на примере. Предположим, что мы отлаживаем эту функцию:

```
function getIterator(countBy, startAt, upTill) {
    countBy = countBy || 1;
    startAt = startAt || 0;
    upTill = upTill || 100;
    var current = startAt;
    return function() {
        current += countBy;
        return (current > upTill) ? NaN : current;
    };
}
```

Данная простая функция создает итератор. Следующий код демонстрирует его использование:

```
var q = getIterator(10, 0, 200);
console.log(q());
```

Теперь отладим эту функцию в Firebug и взглянем на стек, как показано на рис. 7.4. На панели **Стек вызовов** появится функция с именем **(?)**, то есть по сути анонимная. Так вот, чтобы задать имя для анонимной функции, вы можете использовать свойство `displayName` анонимной функции:

```
function getIterator(countBy, startAt, upTill) {
    countBy = countBy || 1;
    startAt = startAt || 0;
    upTill = upTill || 100;
    var current = startAt
        , ret = function() {
```

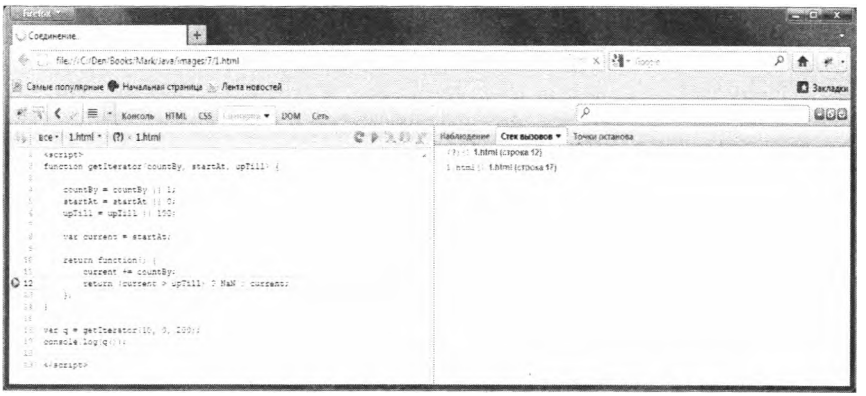


Рис. 7.4. Анонимные функции в Firebug

```
        current += countBy;
        return (current > upTill) ? NaN : current;
    }
;
    ret.displayName = "Iterator от " + startAt + " до "
     + upTill + " by " + countBy;
    return ret;
}
```

Здесь я добавил свойство `displayName` к анонимной функции, благодаря чему информация на панели **Стек вызовов** стала понятнее, см. рис. 7.5. Вместо `(?)` мы получили описывающее имя, да еще и с параметрами функции. Теперь мы знаем, что есть что.

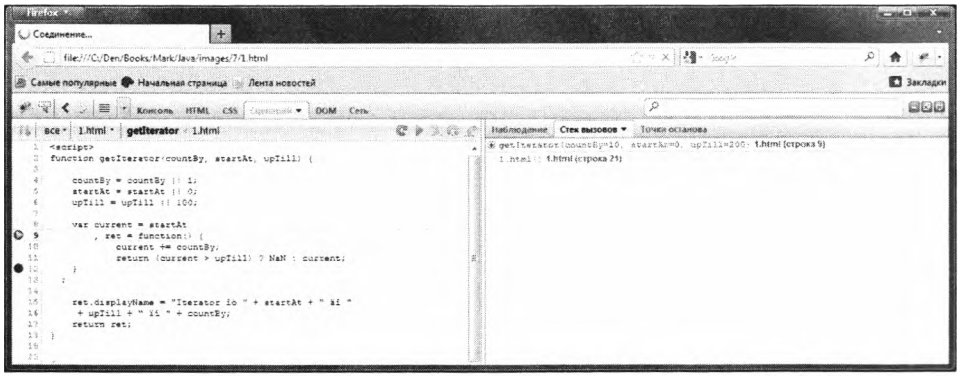


Рис. 7.5. Анонимная функция со свойством diplayName в Firebug

Отладчик Chrome

У Chrome есть аналогичный набор инструментов разработчика (https://developers.google.com/chrome-developer-tools/?hl=ru), получить доступ к которому можно, выбрав в окне браузера Chrome **Инструменты > Инструменты разработчика** или нажав сочетание клавиш Ctrl + Shift + I. Инструменты разработчика могут отображаться в одном окне с отлаживаемой веб-страницей, а могут быть выделены в отдельное окно.

На рис. 7.6 показано, как все это будет выглядеть при изначально открытой в браузере странице Yahoo.com.

Инструменты разработчика Google Chrome содержат ряд вкладок, подобных тем, которые вы видели в Firebug. При этом вклад-

Рис. 7.6. Инструменты разработчика Google Chrome при просмотре Yahoo.com

ка **Timeline** (см. рис. 7.7) предоставляет гораздо больше информации: почти все то же, что вы получаете при использовании утилиты Speed Tracer (см. предыдущую главу), показывая не только сетевой трафик, но и поток пользовательского интерфейса (UI thread), включая прорисовку, события и разметку. График **Timeline** также отображает, когда заканчивается загрузка главного ресурса (фиолетовая линия) и когда заканчивается обработка основного ресурса

Рис. 7.7. Панель Timeline инструментов разработчика Chrome

DOM-документа (красная линия). Первый график показывает, когда пользователь видит начальный HTML, а второй — когда пользователь может взаимодействовать с HTML. Это очень важные графики. На панели **Timeline** также есть график использования памяти. Если данный график всегда восходящий, это может свидетельствовать об утечке памяти — память выделяется, но не освобождается.

Отладчик Chrome поддерживает все консольные методы Firebug, и помимо всего прочего имеет своеобразный «улучшитель», призванный минимизировать сложность восприятия кода и отформатировать его в более читабельный вид. Как только эта функция отформатирует код, вы сможете достаточно легко и удобно установить точки остановки. Когда у вас запутанный и плохо читаемый код эта функция-«улучшитель» особенно полезна.

Отладчик Chrome также поддерживает "именование" блоков `eval`[1], Но если вы не используете такие конструкции, то надеюсь, это на вас никак не скажется. Однако некоторые утилиты, особенно утилиты, манипулирующие JavaScript-кодом, например компилятор CofeeScript, очень активно используют `eval`. Следующий формат:

//@ sourceURL=<любая_строка_без_пробелов>

заставит ваш `eval` появиться в списке **Scripts**, где вы можете выбрать его, чтобы немедленно перейти к оператору `eval`. Рассмотрим небольшой пример:

```
var two = eval('1 + 1; //@ sourceURL=addedOnePlusOne!');
```

Рис. 7.8 показывает, как будет выглядеть панель **Scripts** отладчика Chrome.

Как видно из этого рисунка, `addedOnePlusOne!` доступна в списке сценариев. Щелчок на этом элементе сразу приведет к eval-выражению: вы сможете изучить его и установить точку останова при необходимости.

1 Функция eval (code) позволяет выполнить код, переданный ей в виде строки

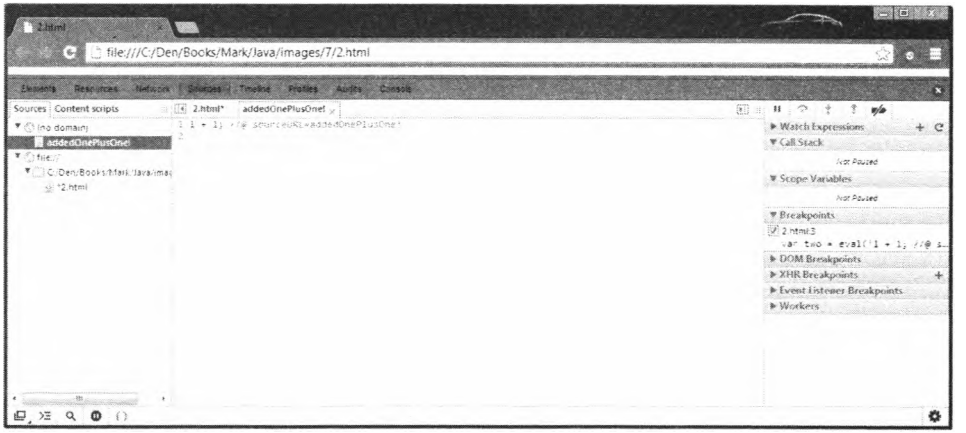


Рис. 7.8. Установка sourceURL в инструментах разработчика Chrome

Браузер Chrome также позволяет легко изменять строку **User Agent**, отправляемую браузером при получении веб-страниц. Получить доступ к этой возможности можно в меню **Настройки**, где вы можете установить любое значение этой строки (рис. 7.9).

Примечание

Чтобы изменить **User Agent** выполните следующие действия: откройте инструменты разработчика, нажав F12; нажмите кнопку-шестеренку в нижнем правом углу экрана; перейдите в раздел **Overrides**; включите параметр **User Agent** и выберите из списка другой браузер и его версию.

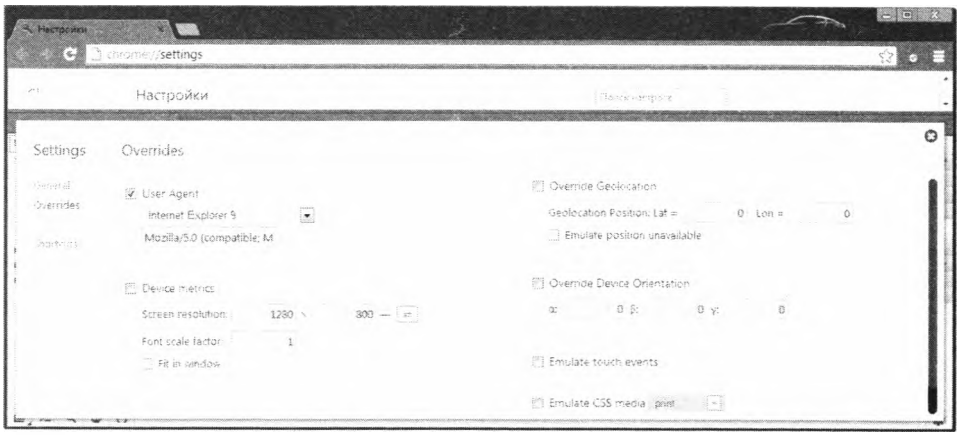


Рис. 7.9. Изменение строки User Agent в Chrome

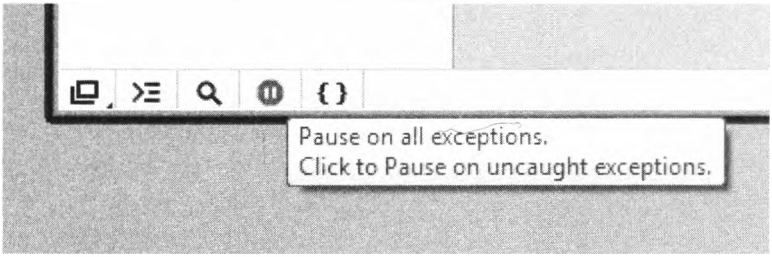


Рис. 7.10. Переключатель «Pause on exceptions» в инструментах разработчика Chrome

В завершение нужно отметить, что Chrome позволяет вам останавливаться на всех ошибках, даже если ошибки были обработаны через блок `try/catch`, что и показано на рис. 7.10.

Вы можете включить и выключить эту функцию. Я же рекомендую оставить ее включенной, чтобы быть уверенными, что вы видите все ошибки и исключения, происходящие в вашей программе.

Отладчик Safari

Так же как и Chrome, браузер Safari основан на WebKit, и поэтому Safari-отладчик очень похож на отладчик Chrome. Чтобы воспользоваться инструментами разработчика в Safari, нажмите кнопку с шестеренкой, выберите команду **Настройки**, в появившемся окне перейдите в раздел **Дополнения** и включите переключатель **Показывать меню «Разработка» в строке меню**. Будет отображено меню, в котором будет пункт **«Разработка»** (см. рис. 7.11) со множеством полезных команд.

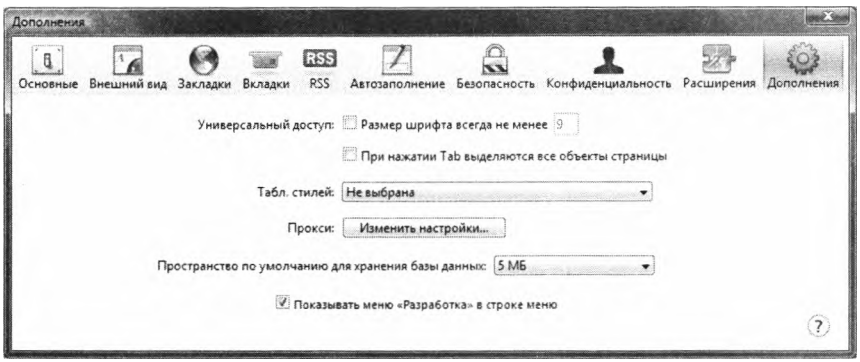


Рис. 7.11. Включение меню «Разработка» в Safari

Рис. 7.12. Веб-инспектор в Safari

Откройте меню **Разработка** и выберите команду **Показать веб-инспектор**. На экране появится **Панель веб-инспектора**, очень похожая на аналогичную панель в Chrome (см. рис. 7.12).

В качестве отличий отладчиков Safari и Chrome стоит отметить, что Chrome добавил некоторые функции, недоступные в стандартном WebKit, а именно — удаленную отладку протокола JSON (даже при том, что эта функция была портирована в сам WebKit, она все еще не доступна в Safari). Также в Chrome есть некоторые другие расширенные настройки. Но в остальном Safari и Chrome ведут себя, в принципе, одинаково. Поэтому для отладки вы можете использовать любой из этих браузеров, какой вам больше нравится. Конечно, между этими двумя браузерами разница есть, но она не такая глобальная, как между Safari/Chrome и Firefox или между Safari/Chrome и Internet Explorer.

Так же как и Chrome, браузер Safari позволяет легко изменить значение строки **User Agent**, что в свою очередь позволяет вам "маскироваться" под другие браузеры: установленное вами значение (наименование браузера) будет отправляться вместо истинного значения **User Agent** на веб-сервер. Этот трюк используется для получения контента, оптимизированного для других браузеров. Особенно

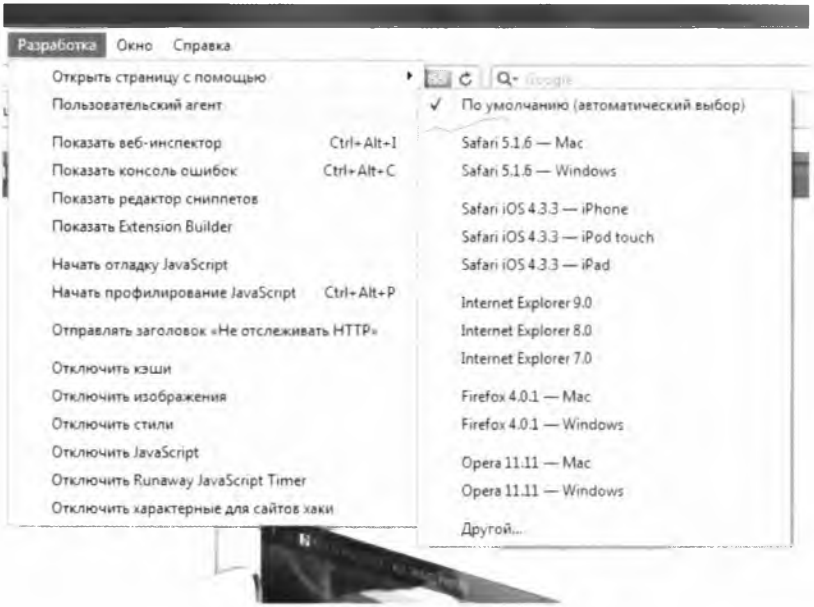


Рис. 7.13. Изменение пользовательского агента в Safari

полезно это бывает тогда, когда вы хотите «закосить» под мобильный браузер и посмотреть, какой контент при этом будет показан. На рис. 7.13 показано, как изменить строку User Agent.

Safari также поддерживает свойство `displayName` для отображения имен анонимных функций в отладчике.

Отладчик Internet Explorer и Edge

К счастью, использование IE версии 8 и более старых уменьшается с каждым днем. Начиная же с версии 8 в IE появился хороший отладчик, доступ к которому можно получить, нажав клавишу F12 (отсюда и название «Средства разработчика F12») или через меню **Сервис - Средства разработчика**. Отладчик IE предоставляет средства, подобные средствам Firefox или WebKit, в том числе есть базовая информация о профилировании.

IE также предоставляет объект console, но с очень ограниченной функциональностью: поддерживаются только методы `log`, `info`, `warn`, `error`, `assert`.

7.2. Отладка Node.js

Node.js поставляется вместе с отладчиком командной строки (http://nodejs.org/api/debugger.html), запустить который можно с помощью команды:

```
% node debug myscript.js
```

У отладчика есть набор небольших команд, просмотреть который можно, введя `help`:

```
% node debug chrome.js
< debugger listening on port 5858
connecting... ok
break in chrome.js:1
 1 var webdriverjs = require("webdriverjs")
 2 , fs = require('fs')
 3 , WebSocket = require('faye-websocket')
debug> help
Commands: run (r), cont (c), next (n), step (s), out (o),
backtrace (bt), setBreakpoint (sb), clearBreakpoint (cb),
watch, unwatch, watchers, repl, restart, kill, list, scripts,
breakpoints, version
debug>
```

Все эти команды тщательно документированы, но назначение базовых команд следующее: запуск, остановка, пошаговое выполнение кода, установка и удаление точек останова и просмотр переменных.

Довольно интересна команда `repl`. Она переключает отладчик в режим REPL (read-evaluate-print-loop), в котором вы можете выполнить с помощью `eval` любой JavaScript-код (в том числе и анализ переменных) в текущем контексте и области, в которой отладчик находится в тот момент. Другими словами, в этом режиме вы можете вводить любой код, например код, позволяющий выводить на консоль значение переменных, которые вы хотите просмотреть. Этот код будет передан функции `eval()` для выполнения. Выход из режима REPL возвратит вас в обычный режим отладчика, в котором вы можете вводить команды, перечисленные ранее.

Вы также можете использовать знакомый оператор `debugger`, позволяющий устанавливать точку останова.

Команда `watch` не очень часто используется, но довольно интересна, поскольку с помощью нее вы можете "наблюдать" за за той или иной переменной. Значения переменных, за которыми вы наблюдаете, будут выведены не только в каждой точке остановки, но и после каждого шага при пошаговом выполнении программы (после команд step, next, или out). При этом в настоящее время условные точки останова не поддерживаются.

Запуск команды `node debug <сценарий>` для интерактивной отладки вашего Node.js сценария бывает очень полезен. V8 — движок, на котором работает Node.js, предоставляет JSON-ориентированный протокол для отладки кода. Для получения доступа к этому режиму используйте аргументы `--debug` и `--debug-brk`. В настоящее время только один отладчик может быть подключен к сеансу отладки за один раз. Запуск `node debug` немедленно использует то единственное соединение отладчика. Флаги `--debug` и `--debug-brk` запускают ваш сценарий Node.js, после чего происходит или использование того самого единственного соединения с удаленным отладчиком, или ожидание.

Подробности протокола отладки V8 JSON доступны по адресу: **https://code.google.com/p/v8/wiki/DebuggerProtocol.** Чтобы использовать его, просто подключитесь к порту отладчика (по умолчанию 5858) и передавайте и принимайте JSON-объекты.

По адресу https://github.com/zzo/pDebug/ доступна базовая реализация протокола. Вы можете установить ее через **npm**:

```
% npm install pDebug -g
```

Теперь вы можете писать программы для отладки и тестирования ваших приложений. Первым делом нужно указать программу, которую вы хотите отладить или исследовать:

```
% node -debug-brk myProgram.js
```

Эта команда загрузит отладчик, но не запустит вашу программу. Подключиться к программе и запустить ее можно с помощью следующих команд отладчика:

```
var pDebug = require('pDebug').pDebug
        , debug = new pDebug({ eventHandler: function(event) {
                console.log('Event'); console.log(event); } })
;
```

```
debug.connect(function() {
        var msg = { command: 'continue' };
        debug.send(msg, function(req, resp) {
                console.log('REQ: ');
                console.log(req);
                console.log('RES: ');
                console.log(resp);
        });
});
```

Вы также можете запустить вашу программу и присоединиться к ней позже, используя следующую команду:

```
% node -debug myProgram.js
```

Используя API, предоставляемый движком V8, вы можете программно отправлять запросы и получать ответы от локального или удаленного отладчика.

Используя npm-модуль `node-inspector()`, вы можете использовать отладчик WebKit и средства разработчика для отладки ваших Node.js приложений:

```
% npm install node-inspector -g
```

Теперь вы можете загрузить ваше Node.js-приложение в отладчике:

```
% node -debug-brk myProgram.js
```

Запустите инспектор в фоновом режиме:

```
% node-inspector &
```

После этого откройте Chrome или Safari и откройте следующий URL: http://localhost:8080/debug?port=5858. Как будто по волшебству откроется отладчик и вы сможете отладить свое Node.js-приложение! Очень, очень удобно! Поскольку PhantomJS основан на WebKit, вы можете передать ему этот URL и открыть консоль отладчика в PhantomJS. На рис. 7.14 изображен снимок экрана PhantomJS, в котором выполняется отладчик WebKit.

Но и это еще не все. Вы можете управлять отладчиком удаленно! Об этом мы и поговорим в следующем разделе. А в завершении этого раздела еще раз подчеркнем важность и удобство использования браузеров Chrome или Safari для визуальной отладки ваших Node.js-приложений.

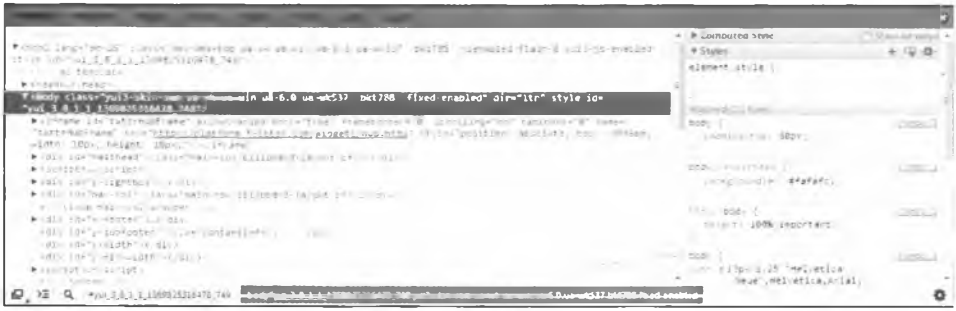


Рис. 7.14. PhantomJS, в котором выполняется отладчик WebKit

7.3. Удаленная отладка

Удаленная отладка JavaScript-кода очень проста и гораздо полезнее, чем вы можете себе даже представить. Пригодится она в следующих случаях:

- Отладка мобильных браузеров.
- Отладка серверных сценариев JavaScript.
- Отладка утилит браузера.
- Программируемая отладка.

Сперва мы вкратце взглянем на особенности удаленной отладки в двух окружениях (Chrome и Firefox), а затем мы рассмотрим каждый сценарий отдельно, чтобы разобраться, как он работает. Удаленная отладка позволяет отладчику запускаться в отдельном процессе или даже на отдельном узле (хосте).

Удаленная отладка в окружении Chrome

Самая зрелая среда удаленной разработки, безусловно, имеется в браузере Chrome. Начиная с версии 31, в Chrome есть последняя версия JSON-ориентированного API удаленной отладки — версия 1.1 (https://developer.chrome.com/devtools/docs/debugger-protocol). Обратите внимание, что это не API отладчика V8, а совсем другой API! Браузер Chrome позволяет использовать свой отладчик на

удаленном коде и позволяет удаленному отладчику подключаться к Chrome для отладки выполняющегося в нем JavaScript-кода.

Для удаленной отладки выполняющегося в браузере JavaScript-кода вам нужно запустить исполнимый файл Chrome, указав номер порта, который будет прослушиваться отладчиком. В Windows команда будет следующей:

```
% chrome.exe --remote-debugging-port=9222
```

В MacOS команда будет немного другой (все в одну строку):

```
% /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --remote-debugging-port=9222
```

Вы также можете запустить Chrome с таким параметром:

```
--user-data-dir=<some directory>
```

Это будет эквивалентно запуску Chrome с новым особым профилем, например, с вашим профилем Firefox. Теперь вы можете присоединиться к порту 9222 и взаимодействовать с профилем отладчика 1.1.

Вот пример того, как использовать эту функциональность, чтобы извлечь статистику таймингов — эту статистику визуализируют Speed Tracer и панель **Timeline** отладчика Chrome. Мы будем использовать Selenium, чтобы автоматически породить экземпляр Chrome, слушающий порт 9222 для соединений отладчика. Интересно, что JSON-отладка API работает на основе веб-сокетов, а не на основе HTTP. Использование веб-сокетов делает распространение события более тривиальным. Использование `EventHub` или `socket.io` предоставляет ту же функциональность по обычному HTTP.

Теперь давайте взглянем на сценарий Node.js, использующий пакет **webdriversjs** (см. гл. 6). Этот сценарий довольно длинный, поэтому мы разобьем его на две части. Первая часть запускает Selenium с соответствующими опциями командной строки Chrome и выполняет браузер с потоком, который мы хотим захватить:

```
var webdriverjs = require("webdriverjs")
```

```
    , browser = webdriverjs.remote({
        host: 'localhost'
        , port: 4444
        , desiredCapabilities: {
            browserName: 'chrome'
            , seleniumProtocol: 'WebDriver'
            , 'chrome.switches': [
                '--remote-debugging-port=9222'
                , '--user-data-dir=remote-profile'
              ]
        }
    }
);
browser.addCommand("startCapture", startCapture);
browser
    .init()
    .url('http://search.yahoo.com')
    .startCapture()
    .setValue("#yschsp", "JavaScript")
    .submitForm("#sf")
    .saveScreenshot('results.png')
    .end();
```

Все это — стандартный код **webdriverjs**, с которым мы знакомились чуть ранее. Однако, к нему есть два дополнения. Первое дополнение — желаемая возможность `chrome.switches` (предоставляет доступ к параметрам командной строки Chrome, по сути является массивом параметров). Драйвер Chrome позволяет нам указывать параметры командной строки для исполнимого файла Chrome. И мы при вызове браузера говорим Chrome, что нужно слушать удаленные соединения отладчика на порту 9222 и использовать профиль из определенного каталога. Для использования удаленной отладки нам не нужны ни какие-то определенные расширения или настройки Chrome. Поэтому мы просто используем пустой профиль.

Второе дополнение — это функция `startCapture`. Пакет **webdriverjs** позволяет нам определить свой пользовательский метод, который может быть объединен в цепочку вместе с другими стандартными методами **webdriverjs**. Любой пользовательский метод получает функцию обратного вызова, которая будет вызвана по завершении пользовательской функции. Здесь мы определили метод

`startCapture` для «захвата» всей информации по таймингам, которую позволяет предоставить Chrome, включая время передачи по сети, время прорисовки, время разметки и т.д.

Далее приведен код функции `startCapture`:

```
function startCapture(ready) {
  var http = require('http')
    , options = {
    host: 'localhost'
    , port: 9222
    , path: '/json'
   }
;
http.get(options, function(res) {
  res.on('data', function (chunk) {
  var resObj = JSON.parse(chunk);
  connectToDebugger(resObj[0], ready);
 });
}).on('error', function(e) {
 console.log("Got error: " + e.message);
}
```

При посещении *http://localhost:9222/json* будет получен ответ JSON, который является массивом вкладок, которые могут использоваться при удаленной отладке. Так как мы открыли этот экземпляр Chrome с Selenium, у нас есть только одна вкладка, следовательно, будет только один объект в массиве. Обратите внимание на то, что **webdriverjs** передал функции `startCapture` функцию обратного вызова, которая будет вызвана, как только `startCapture` завершит работу. Эта функция подключится к локальному (или удаленному) экземпляру Chrome, получит JSON-данные о текущей вкладке, которую мы хотим отладить, а затем передаст эту информацию функции `connectToDebugger`, приведенной ниже:

```
function connectToDebugger(obj, ready) {
  var fs = require('fs')
    , WebSocket = require('faye-websocket')
    , ws = new WebSocket.Client(obj.webSocketDebuggerUrl)
    , msg = {
      id: 777
      , method: "Timeline.start"
```

```
            , params: {
              maxCallStackDepth: 10
            }
          }
          , messages = ''
;
ws.onopen = function(event) {
  ws.send(JSON.stringify(msg));
  ready();
};
ws.onmessage = function(event) {
  var obj = JSON.parse(event.data);
  if (obj.method && obj.method === 'Timeline.eventRecorded') {
    obj.record = obj.params.record; // Небольшой трюк
    messages += JSON.stringify(obj) + '\n';
  }
};
ws.onclose = function(event) {
  var header = '<html isdump="true">\n<body><span id="info">'
   + '</span>\n<div id="traceData" isRaw="true" version="0.26">'
   , footer = '</div></body></html>'
  ;
  ws = null;
  fs.writeFileSync('DUMP.speedtracer.html', header + messages
   + footer, 'utf8');
};
```

Помним, что протокол отладчика работает по веб-сокетному соединению. При этом в рамках вышеприведенного примера мы используем модуль **faye-websocket**, который обеспечивает превосходную клиентскую реализацию веб-сокетов.

Объект, предоставляемый `startCapture`, содержит свойство `webSocketDebuggerUrl`, являющееся веб-адресом сокета. После соединения с указанным адресом веб-сокета, модуль faye-websocket выпустит обратный вызов к функции `onopen`. А наша функция `onopen` отправит одно JSON-закодированное сообщение:

```
msg = {
  id: 777
  , method: "Timeline.start"
  , params: {
```

```
        maxCallStackDepth: 10
    }
);
```

Данное сообщение указывает браузеру Chrome отправлять нам сообщения веб-сокета, которые являются событиями отладчика Chrome, то есть отправлять все события, которые изначально предназначены для отображения на панели Timeline. Как только мы отправили вышеуказанное сообщение, мы вызываем обратный вызов **webdriverjs** так, что наша цепочка **webdriverjs**[2] может быть продолжена (помните это?).

Примечание.

Спецификация для вышеприведенного JSON-сообщения доступна на сайте инструментов разработчика Chrome (https://developer.chrome.com/devtools/docs/debugger-protocol).

Теперь нам остается поделать какие-то действия в Selenium, для которых мы хотим получить события **Timeline** — в данном случае мы загрузим Yahoo! и поищем строку «JavaScript». Пока все это происходит, Chrome отправляет нам события `Timeline.eventRecorded`, которые мы покорно получаем и сохраняем через обратный вызов веб-сокета **onmessage**.

Наконец, по завершению всех этих действий, мы закрываем Selenium, который закроет Chrome, а он, в свою очередь, вызовет наш обработчик `onclose`, который выведет все наши сохраненные

---

2 Пакет **webdriverjs** позволяет нам определить свой пользовательский метод, который может быть объединен в цепочку вместе с другими стандартными методами **webdriverjs** события в HTML-файл. Формат этого файла такой же, какой использует рассмотренный нами ранее Speed Tracer, что очень удобно. Поэтому если вы загрузите получившийся HTML-файл в Chrome с установленным расширением Speed Tracer, тот загрузит интерфейс Speed Trace с этими данными. Это позволяет вам по сути «захватывать» ввод Speed Tracer, что можно использовать для сравнения с предыдущими версиями вашего кода или исследовать его на досуге.

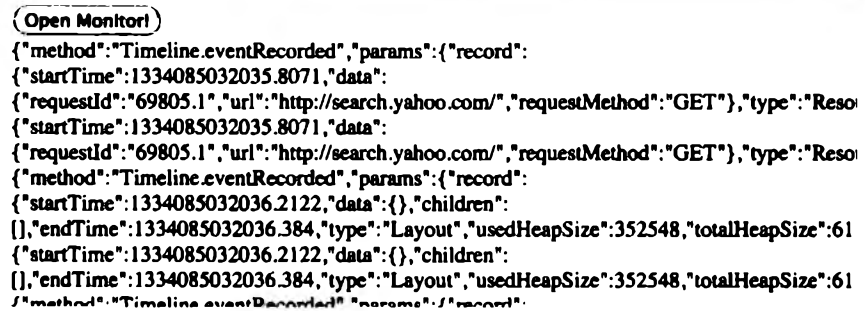


Рис. 7.15. HTML-код после загрузки в Chrome с установленным расширением Speed Tracer

На рис. 7.15 показано, как выглядит сгенерированный HTML-код, когда он загружен в Chrome с установленным Speed Tracer. Нажатие кнопки **Open Monitor!** откроет интерфейс Speed Tracer с нашими захваченными данными в нем, как показано на рис. 7.16.

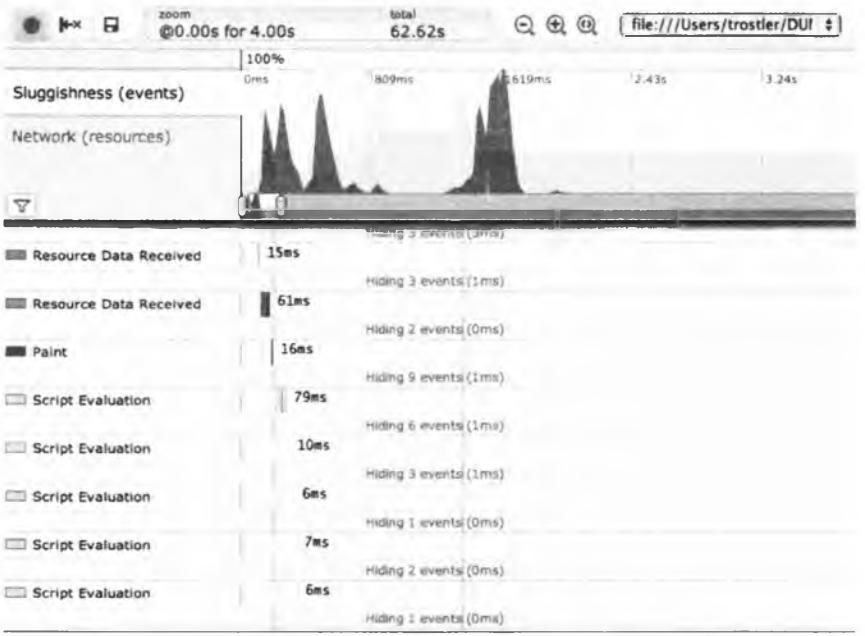


Рис. 7.16. Speed Tracer показывает захваченные данные

Таким образом, мы теперь с вами научились полностью управлять отладчиком Chrome из удаленного расположения, приостанавливать и возобновлять сценарии, получать консольный вывод и делать многие другие полезные вещи.

Отладка в PhantomJS

Аналогично Chrome, PhantomJS тоже может использоваться для удаленной отладки. Но в этом случае удаленная машина может быть бездисплейной или вы можете запустить PhantomJS локально без необходимости запускать отдельный экземпляр браузера с его собственным отдельным профилем.

К сожалению, вы не сможете непосредственно взаимодействовать с веб-приложением, запущенным в браузере PhantomJS. Однако, вы сможете перемещаться по коду приложения и взаимодействовать с ним программно с помощью PhantomJS-сценария. Таким образом, удаленная отладка с использованием PhantomJS наиболее полезна для программируемой отладки.

Ниже приведена команда для настройки последней версии PhantomJS (2.0 на момент написания этих строк):

```
% phantomjs -remote-debugger-port=9000 ./loader.js <webapp URL>
```

В данной команде `loader.js` — это сценарий PhantomJS, который просто загружает ваш URL и ждет:

```
// Открываем страницу из командной строки PhantomJS
var page = new WebPage();
page.onError = function (msg, trace) {
  console.log(msg);
  trace.forEach(function(item) {
    console.log('  ', item.file, ':', item.line);
  })
}
page.open(phantom.args[0], function (status) {
  // Проверяем, успешно ли загружена страница
  if (status !== "success") {
    console.log("Нет доступа к сети");
  } else {
    setInterval(function() {}, 200000);
```

```
    }
});
```

Вышеприведенный сценарий запишет любые синтаксические ошибки или непойманные исключения из вашего приложения, что есть хорошо. Во время работы этого сценарий откройте свой локальный WebKit-браузер и в строке адреса введите *http://<phantomjs_host>:9000*. Откроется знакомый отладчик и вы сможете заняться отладкой вашего приложения «вживую».

PhantomJS также предоставляет метод `evalutate`, позволяющий вам управлять DOM загруженной страницы и выполнять JavaScript на этой странице. Обратите внимание на то, что любой JavaScript, выполняемый PhantomJS на странице, работает в «песочнице» и не может взаимодействовать с рабочим JavaScript в вашем приложении, но именно для этого и нужен удаленный отладчик! Одностороннее взаимодействие вашего приложения с PhantomJS может осуществляться через сообщения `console.log`.

Отладка в Firefox

Расширение Firebug также обладает возможностями удаленной отладки, используя расширение Crossfire (**http://getfirebug.com/wiki/index.php/Crossfire**). Crossfire — это расширение Firebug, которое представляет протокол сетевой программной отладки. Последняя версия (0.3a10 на данный момент) доступна на сайте: **http://getfirebug.com/releases/crossfire/**. Установите Crossfire, перезапустите Firefox и вы увидите значок **Crossfire** на панели расширений. Нажмите этот значок, а затем нажмите кнопку **Start Server** и вы увидите диалоговое окно, показанное на рис. 7.17.

Нажмите **OK** для запуска сервера Crossfire на этом узле. По умолчанию используется порт 5000 (при необходимости вы можете изменить номер порта).

В качестве альтернативы вы можете запустить Firefox с параметром командной строки для автоматического запуска сервера Crossfire на заданном порту:

```
% firefox -crossfire-server-port 5000
```

После этого вы можете подключиться к серверу Crossfire и начать отладку. Подобно вкладкам Chrome, Firebug обрабатывает каждую

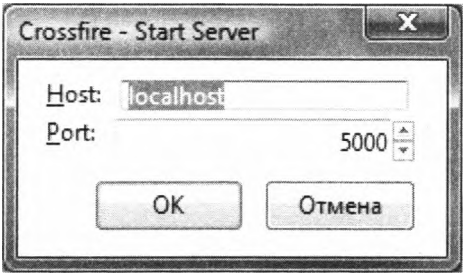


Рис. 7.17. Диалог конфигурации Crossfire в Firefox

вкладку как отдельный «контекст». К сожалению, драйвер Firefox в Selenium в настоящее время не поддерживает передачу произвольных параметров командной строки исполнимому файлу Firefox и поэтому Crossfire не поддерживает основанную на профиле конфигурацию. Таким образом, автоматизация средствами Selenium невозможна. Но надеюсь, что это скоро изменится.

7.4. Мобильная отладка

Мобильная отладка – это по сути просто ответвление удаленной отладки. Чтобы быть предельно точным, мы сейчас говорим об отладке в мобильных браузерах. В рамках данной книги для мобильной отладки мы будем использовать последние версии Android (4.2, 4,4 и 5) и iOS 7/8, а также WebKit-браузеры: Chrome в Android и Safari в iOS.

Android

Установить Chrome для Android можно из Google Play (https://play.google.com/store/apps/details?id=com.android.chrome&hl=ru). Данная версия поддерживает удаленную отладку с вашего настольного браузера с использованием стандартного протокола отправки WebKit. Как только Chrome для Android будет установлен, перейдите в **Settings>Developer Tools** для включения удаленной отладки.

Тем временем, на настольной системе вам нужно установить Android Software Developer Kit (SDK), который можно скачать по адресу http://developer.android.com/sdk/index.html. Со всего этого пакета нам нужен только один сценарий: Android Debug Bridge (adb). Этот исполнимый файл находится в стандартной загрузке

SDK и установить его несложно: как только вы загрузили SDK, запустите Android SDK Manager. С помощью данной утилиты установите утилиты платформы (Android SDK Platform tools), которые и содержат утилиту **adb**. Просто установите соответствующий флажок и нажмите кнопку **Install**. На рис. 7.18 показан Android SDK Manager на моем Mac.

Установка всего, что предлагает установить Android SDK Manager, займет очень много времени, поэтому если вам нужен только adb, отключите все остальные флажки и нажмите кнопку **Install**. Как только инструмент будет установлен, в основном дереве SDK появится каталог `platform-tools`. Теперь вам нужно подключить ваш телефон к вашему компьютеру по USB. После чего запустите adb следующим образом:

```
% ./adb forward tcp:9222 localabstract:chrome_devtools_remote
```

На вашем настольном Chrome подключитесь к http://localhost:9222 и вы увидите список URL на каждой вкладке браузера Chrome на вашем телефоне. Щелкните по вкладке, которую нужно отладить и

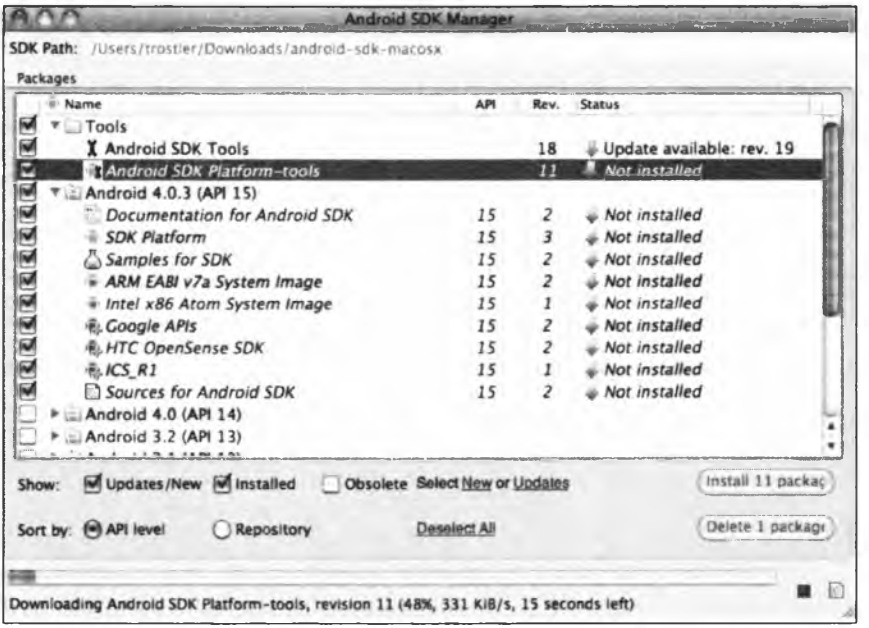


Рис. 7.18. Android SDK Manager

после этого вы сможете использовать инструменты разработчика на своем настольном Chrome для отладки и управления браузером вашего телефона.

Самый большой недостаток этого способа — вы должны использовать Android 4.0 или более позднюю версию, браузер Chrome Beta и ваш телефон должен быть физически соединен с вашим настольным компьютером.

iOS

Еще в iOS 6 появилась новая функция удаленного Web Inspector в Mobile Safari. Чтобы активировать ее на вашем iDevice выберите **Settings->Safari->Advanced** (см. рис. 7.19). Теперь откройте сайт, который вы хотите отладить в Mobile Safari, подключите устройство к Mac, на котором запущена настольная версия Safari и разверните меню **Develop (Разработка)**, как показано на рис. 7.20 (меню "Разработка" должно быть включено).

Рис. 7.19. Включение удаленного Web Inspector в iOS

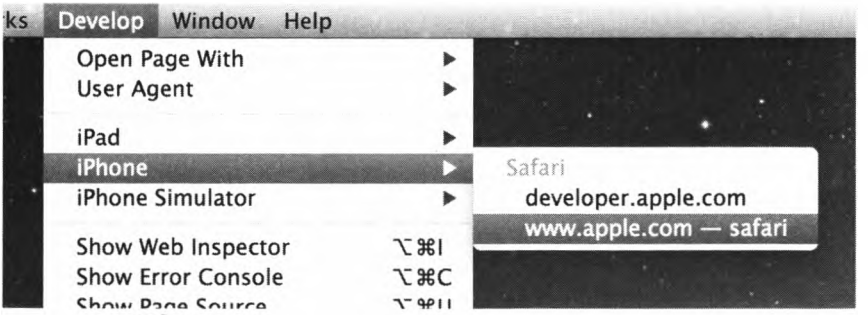


Рис. 7.20. Отладка сайта на iPhone из настольного Safari

Как видите, мой iPhone теперь доступен для удаленной отладки. Вы можете удаленно подключиться к любой странице и затем использовать отладчик обычным образом: устанавливать точки останова, пошагово выполнять JavaScript, просматривать элементы страницы и т.д.

На рис. 7.21 изображено окно Safari Web Inspector при отладке страницы.

Кроссплатформенное решение Adobe Edge Inspect

Рис. 7.21. Веб-инспектор Safari при отладке мобильной версии страницы Yahoo.com

Adobe Edge Inspect (исходно Adobe Shadow) — кросс-платформенное решение, позволяющее производить удаленную мобильную WebKit-отладку на всех версиях Android, iOS и Kindle Fire. Состоит это решение из расширения для настольной версии Chrome, настольного демона (подобного adb) и специального приложения на вашем мобильном устройстве. Ваше устройство должно находиться в одной локальной сети, что и ваш стационарный компьютер (или ноутбук) — так, чтобы они могли взаимодействовать друг с другом. Adobe Edge Inspect работает только на Windows 7/8/10 или Mac OS X.

Перейдите на сайт Adobe Creative Cloud (http://html.adobe.com/edge/inspect/) и получите последнюю версию демона Inspect. По адресу https://chrome.google.com/webstore/category/apps можно поискать и скачать расширение Inspect для браузера Chrome. После установки расширения в Chrome вы увидите пиктограмму Inspect в вашем браузере. Нажмите ее и вы увидите экран наподобие изображенному на рис. 7.22. В любом случае, для работы этого расширения нужно сперва установить программу Adobe Edge CC Inspect. Если она не установлена, вам будет выдано соответствующее предупреждение.

Убедитесь, что переключатель (см. рис. 7.22) переведен в состояние "on" (включено). Теперь расширение Adobe Inspect готово прини-

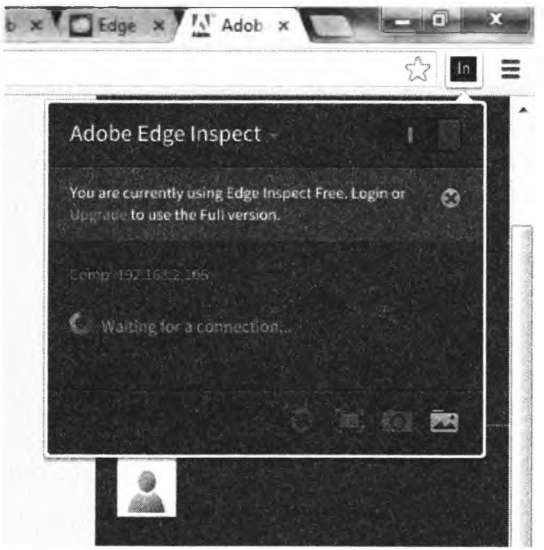


Рис. 7.22. Расширение Adobe Inspect в Chrome

мать соединения с мобильного устройства, на котором также запущено приложение Adobe Edge Inspect, поэтому, соответственно, вам нужно скачать его для вашего мобильного устройства и запустить. Вас попросят добавить подключение. Хотя доступна функция "обнаружения" устройств, иногда она не работает. Поэтому, возможно, вам придется вручную добавить соединение, указав в мобильном устройсте IP-адрес вашего стационарного компьютера, на котором запущен демон Adobe Inspect (см. рис. 7.23). После соединения программа отобразит поле ввода PIN-кода, который нужно ввести в расширении Adobe Inspect, запущенном в вашем "настольном" браузере Chrome (см. рис. 7.24). После этого устройство полностью готово к отладке (рис. 7.25).

Теперь откройте пустую вкладку в вашем настольном браузере Chrome и перейдите на страницу, которую вы хотите отладить. Эта страница также будет загружена на вашем мобильном устройстве. Фактически, как только вы подключились, любой посещенный вами URL будет дублироваться на подключенном устройстве. Нажатие пиктограммы <> в вашем расширении Inspect запустит отладчик, который займется отладкой открытой страницы на мобильном устройстве.

Рис. 7.23. Укажите IP-адрес настольного компьютера на iPhone

Рис. 7.24. Введите PIN на настольном Chrome

Теперь вы можете использовать отладчик, как будто отлаживаете локальную страницу: вы можете выбирать элементы, изменять CSS, отлаживать JavaScript-код.

Это работает во всех версиях Android, iOS и Kindle Fire. Однако есть следующие требования: компьютер и мобильное устройство должны быть в одной локальной сети, вы должны использовать Chrome, который должен работать под управлением Windows 7/8 или Mac OS X.

Рис. 7.25. Отладка удаленного устройства

Когда сайт предлагает и "мобильную", и "настольную" версии отображения своего содержимого, то в качестве значения строки User Agent на вашем настольном браузере Chrome обязательно нужно установить так, чтобы она соответствовала вашему мобильному устройству. Для этого, находясь в Инструментах разработчика **(Developer Tools)** в нижнем правом углу нажмите шестеренку и выберите вкладку **Overrides** (рис. 7.26). В результате в настольком Chrome сайт станет выглядеть так, как он выглядит на мобильном устройстве.

В завершение этого небольшого раздела хотелось бы отметить, что Adobe Edge CC Inspect имеет целый ряд ограничений, потому лич-

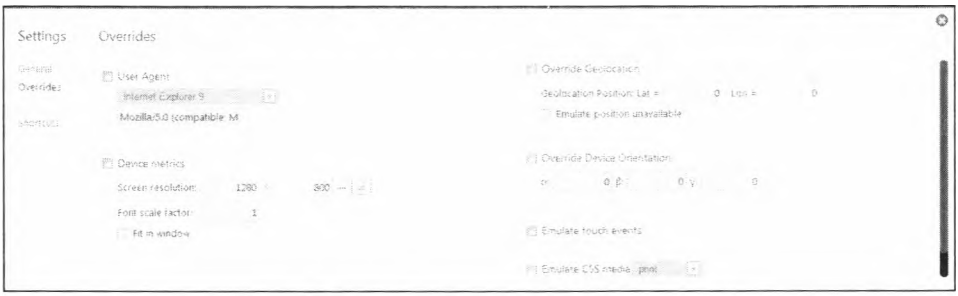


Рис. 7.26. Изменение User Agent в Chrome

но я рекомендую использовать для отладки родные отладчики iOS и Android.

Другие возможности мобильной отладки. Windows Phone

В качестве некоторой альтернативы, для расширения кругозора, давайте кратко рассмотрим еще несколько других возможностей мобильной отладки. Начнем с такого решения, как **weinre**, который стал частью PhoneGap и теперь является проектом Apache Cordova **(http://cordova.apache.org/)**. Средство **weinre (http://people.apache.org/~pmuellr/weinre/docs/latest/)** означает WEb INspector REmote и работает примерно так же, как Adobe Inspect, просто Inspect более популярен.

В качестве еще одного инструмента отладки отметим ресурс **JSconsole.com**, требующий вставки JavaScript-кода для отладки. Конечный результат аналогичен результату использования вышеописанных инструментов, но этот инструмент более неуклюж и сложен.

Вообще, лучше всего использовать родные возможности отладки, рассмотренные в этой книге, или же использовать Adobe Inspect. Но все вокруг очень быстро изменяется, поэтому будьте начеку — вдруг появится какой-то новый и более совершенный инструмент для мобильной отладки.

Кстати, **weinre** с января 2015 года поддерживает отладку и под Windows Phone.

7.5. Производственная отладка

Методы и инструменты, рассмотренные нами до этого, отлично справляются с отладкой в вашем локальном окружении отладки. Но что делать с кодом, который уже работает, который, как говорится «в продакшене»? Минификация, объединение и путаница производственного кода существенно усложняют его отладку. Или что насчет JavaScript-кода, сгенерированного другим инструментом, с другого языка программирования, с такого как CoffeeScript или даже Java (через GWT)?

Минификация и деминификация кода

Минификация кода – это своего рода оптимизация кода, которая заключается в удалении из него всего лишнего: пробелов, табуляций, каких-то ненужных символов, зачастую комментариев и проч. Сам код по сути при этом не меняется, но его объем может быть существенно уменьшен.

Главный недостаток такой оптимизации состоит в том, что код как правило приводится к абсолютно не читабельному виду. Поддержка, редактирование и отладка такого кода весьма затруднительна. Так, рис. 7.27 показывает минифицированный код в отладчике Chrome.

```
1 (function(){var g=void 0,h=!0,i=null,j=!1,ba=encodeURIComponen
2 function ya(a){var b=1,c=0,d;if(!C(a)){b=0;for(d=a[u]-1;0<=d;d-
3 var Ba=function(a,b,c,d){a.addEventListener?a.addEventListener
4 Ea[v].contains=function(a){return this.get(a)!==g};function Fa
5 function Ha(a,b){function c(b,c){a.contains(b)||a.set(b,[]);a.
6 var Pa=K(),Qa=K(),Ra=K(),Sa=K(),Ta=K(),L=K(),M=K(),Ua=K(),Va=K
7 Zb=K(h),$b=K(h),ac=K(h),bc=K(h),cc=K(h),dc=Oa("campaignParams"
8 nb,63,g,1);U("_setLocalRemoteServerMode",nb,47,g,2);U("_setSam
9 Wa,29,1);U("_setReferrerOverride",xb,49);U("_setSiteSpeedSampl
10 21);a("_addItem",S[v].O,19);a("_setTransactionDelim",S[v].la,8
11 14);a("_clearOrganic",S[v].T,70);a("_cookiePathCopy",S[v].W,30
12 Aa,81);a("_setHrefExamineLimit",Aa,80)},R=function(a,b,c,d){a[
13 function(c,d,e){Na[c]&&this[ia]();e?b[c]=d:a[c]=d;Na[c]&&this.
14 var Ec=function(a){var b=this;this.l=0;var c=a.get(fc);this.Da
15 var Kc=function(a,b,c){c=c?"":a.c(L,"1");b=b[w](".");if(6!==b[
16 a.b(Rb,0)][z](".")},Nc=function(a,b,c){var c=c?"":a.c(L,"1"),d
17 0<b[u]&&a.set(Hb,F(b[0]));if(1>=b[u])return h;b=b[1][w](-1==b[
18 b(Xb,"utmccn");b(ac,"utmcmd");b(bc,"utmctr");b(cc,"utmcct");re
19 a.set(b,c))}-1==b[p]("=")&&(b=F(b));var e="2"==c("utmcvr");d(Wi
20 a=a+"="+b+"; path="+c+"; ";e&&(a+="expires="+(new Date((new Da
21 a.language+b.platform+b.userAgent+a.javaEnabled+a.I+a.H+(H.cool
22 else{d=d+"."+d;try{c=new ActiveXObject(d+".7"),e=c.GetVariable
23 jd[ka](Jb),tc=J[sc],rc+=g!=tc?tc:sc;f+=rc;c[m](f)}b+=o+c[z](s)-
24 c,d){if(!hd(d))return j;a(b,"v",c,d[t]());return h};e.getKey=f
25 Pc(a,Ic(b,W("__utmv")));nd=!c;return h},pd=function(a){nd||0<W
26 !C(a.get(Wb))||!C(a.get($b))||!C(a.get(Yb))||!C(a.get(Zb)),c={
27 Fa(d[na]),k=0;k<e[u];++k)if(-1<f[p](e[k])){d=j;break b}zd(a,g,
28 "-",f=Da(b.get(a.get(fb)))||"-",k=Da(b.get("dclid"))||"-",o=c(
29 f,d=a.get(qb),c=F(c)[A](),k=0;k<d[u];++k)if(c==d[k]){c=h;break
30 f=a.get(f)||"-";if(c(k)!=c(f))return h}return j},Cd=RegExp(/^h
31 c=[J,N];break a}J=J[n](/\+/g,"%20");N=N[n](/\+/g,"%20");if(c==
32 e=b[p]("#");if(c)return 0>e?b+"#"+d:b+"&"+d;c="";f=b[p]("?");0<
33 a.get(sb),d=0;d<c[u];d++)if(c[d].id_==b)return c[d];return i};
```

Рис. 7.27. Минифицированный код в инструментах разработчика Chrome

Как видите, с точки зрения читабельности код просто ужасен! Невозможно установить точку останова, и если код где-то остановится, невозможно сказать, где именно. Однако, как уже было ранее отмечено, у WebKit-браузеров есть кнопка «деминификации» кода. При отладке в браузере Chrome нажатие кнопки **{}** делает его читабельным, что и продемонстрировано на рис. 7.28.

Посмотрев на этот код, мы можем сделать вывод, что теперь он читаем и мы можем установить точки останова в любой строке.

Карты кода

Обратившись к коду, показанному на рис. 7.28, отметим, что он обрабатывался/оптимизировался (в английском языке для этого используется слово obfuscated), но означает это то, что в ходе оптимизации длинные, но понятные имена переменных, функций и проч.,

Рис. 7.28. Расширенный код в инструментах разработчика Chrome

заменяются короткими (что сокращает код), но абстрактными и непонятными именами компилятором Closure Compiler (https://developers.google.com/closure/compiler/), поэтому у нас нет доступа к исходным именам переменных . На помощь приходят так называемые карты кода (source maps).

Карта кода — это один файл, содержащий один большой JSON-объект, который дает точное соответствие элементов готового рабочего кода проекта (после всех оптимизиций, минификаций и проч.) и вашего кода разработки.

Скажем больше: исходный код не обязательно должен быть JavaScript. Это очень приятный бонус!

Примечание.

Спецификации карт кода можно почитать по следующему адресу: https://docs.google.com/document/d/1U1RGAehQwRypUTovF1KRlpiOFze0b-_2gc6fAH0KY0k/edit?hl=en_US&pli=1&pli=1.

Использование карт кода позволяет отобразить производственный код обратно в исходный код в отладчике. В Beta-ветке браузер Chrome поддерживает карты кода "из коробки", чего не скажешь о других браузерах. Компилятор Google Closure Compiler объединяет и минимизирует код, он же может помочь с выводом карты кода для этого кода. Google также предоставляет расширение Closure Inspector, которое позволяет "научить" расширение Firebug понимать карты кода, что позволяет использовать браузер для работы и Firefox.

Давайте на примере рассмотрим процесс использования карт кода. Для этого примера мы будем использовать функцию `getIterator()`, с которой мы уже знакомы. Представим, что она сохранена в файле с именем `iterator.js`:

```
function getIterator(countBy, startAt, upTill) {
    countBy = countBy || 1;
    startAt = startAt || 0;
    upTill = upTill || 100;
    var current = startAt
        , ret = function() {
            current += countBy;
            return (current > upTill) ? NaN : current;
        }
    ;
    ret.displayName = "Inerator from " + startAt + " until "
      + upTill + " by " + countBy;
    return ret;
}
```

Как видите, здесь нет ничего необычного. Данная функция «весит» всего 406 байтов. Теперь получите последнюю версию компилятора `compiler.jar` с домашней страницы компилятора Google Closure Compiler. Запустите Closure Compiler, «скормив» ему наш `iterator.js`:

```
% java -jar compiler.jar --js iterator.js --js_output_file it-comp.js
```

Он создаст новый файл, `it-comp.js`:

```
function getIterator(a,b,c){var a=all1,b=bll0,c=cll100,d=b,e=function(){d+=a;return d>c?NaN:d};
e.displayName="Iterator from "+b+" until "+c+" by "+a;return e};
```

Данный файл занимает всего 160 байтов! В дополнение к удалению всех пробелов (и любых комментариев) имена переменных были изменены: `countBy` было преобразовано в `a`, `startAt` — в `b`, `upTill` — в `c` и т.д. Это чрезвычайно сокращает размер кода, и теперь мы готовы разместить его в производственной среде!

Но у нас есть одна проблема: отладка такого кода в браузере будет очень сложной. Даже использование кнопки **{}** в отладчике WebKit не восстановит код в первоначальном виде (или хотя бы похожем на него), поскольку имена переменных были изменены.

На рис. 7.29 показана производственная версия нашего кода (см. 7.29).

Рис. 7.29. Производственный код итератора

Рис. 7.30. Распакованный код итератора

На рис. 7.30 можно уже наблюдать деминифицированную версию кода нашей функции итератора. Эта версия уже менее уродлива, но имена переменных все еще не соответствуют нашему исходному коду, а номера строк отличаются от номеров строк в исходном файле. Вот тут-то нам и помогут карты кода! Прежде всего, нужно указать Closure Compiler, что нужно создать карту кода (все в одной строке):

```
% java -jar compiler.jar --js iterator.js --js_output_file
it-comp.js --create_source_map ./it-min.js.map --source_map_format=V3
```

В результате мы получим файл с картой источника, *it-min.js.map*, содержащий гигантский JSON-объект:


```
{
"version":3,
"file":"it-comp.js",
"lineCount":1,
"mappings":"AAAAA,QAASA,YAAW,CAACC,CAAD,CAAUC,CAAV,CAAAmBC,CAAnB,CAA2B,
CAE3C,IAAAF,EAAUA,CAAVA,EAAqB,CAArB,CACAC,EAAUA,CAAVA,EAAqB,CADrB
,CAEAC,EAASA,CAATA,EAAmB,GAFnB,CAIIC,EAAUF,CAJd,CAKMG,EAAMA,QAAQ
,EAAG,CACfD,CAAA,EAAWH,CACX,OAAQG,EAAA,CAAUD,CAAV,CAAAoBG,GAApB,CAA0BF
,CAFnB,CAMvBC,EAAAAE,YAAA,CAAkB,gBAAIB,CAAqCL,CAArC,CAA+C,SAA/C,CAA2DC
,CAA3D,CAAoE,MAApE,CAA6EF,CAE7E,OAAOI,EAfoC;",
"sources":["iterator.js"],
"names":["getIterator","countBy","startAt","upTill","current","ret"
,"NaN","displayName"]
}
```


Суть этого объекта – свойство `mappings`, которое является Base64 VLQ-закодированной строкой. При желании, в Интернете вы найдете много информации по этому кодированию.

Последнее действие заключается в том, чтобы связать созданную карту кода с JavaScript-кодом, поданным браузеру. Для этого нужно указать отладчику, где найти соответствующую карту:

```
function getIterator(a,b,c){var a=a||1,b=b||0,c=c||100,d=b
,e=function(){d+=a;return d>c?NaN:d};
e.displayName="Iterator from "+b+" until "+c+" by "+a;return e};

//@ sourceMappingURL=file:////Users/trostler/it-min.js.map
```

Рис. 7.31. Исходный код итератора

К выводу Closure Compiler мы добавили строку, указывающую на размещение карты кода, сгенерированной компилятором.

Теперь давайте вернемся в браузер и посмотрим на отладчик (рис. 7.31). В окне отладчика виден не только наш точный исходный код, но имя исходного файла. Также можно установить точки останова, как будто бы работаете с обычным исходным кодом.

Самая большая проблема при использовании карт кода — это вывод/вычисление выражений. Поскольку браузер выполняет на самом деле минимизированный код, то исходные имена переменных фактически не существуют в браузере. Поэтому вы не можете определить значения переменных и при этом вы не знаете, во что была преобразована та или иная переменная.

В отладчике я смотрю на код, в котором определена переменная `current`, но вкладка **Watch Expressions** сообщает, что такая переменная не определена. С другой стороны, определена некоторая случайная переменная с именем `d`, значение которой 0, но этой переменной не видно в листинге исходного кода.

Компилятор Google GWT также может генерировать карты кода, в том числе и для компилятора CoffeeScript. Поэтому вы можете сказать, что отлаживаете свой собственный исходный код, в то время как браузер фактически выполняет скомпилированную версию.

Глава 8.

Автоматизация и развертывание

Автоматизация призвана максимально ускорить процессы создания и работы с JavaScript-кодом. При этом нужно понимать:

- что автоматизировать,
- когда автоматизировать,
- как автоматизировать.

Обо всем этом мы и поговорим в данной главе.

8.1. Что автоматизировать?

Определить, что автоматизировать просто: нужно автоматизировать почти ВСЕ, включая и сам процесс автоматизации! А именно: повторное использование кода, использование JSLint и другие формы статического анализа, все тесты, получение отчетов о покрытии кода, сборку и развертывание приложения (включая откат), отчеты об ошибках, статистику и т.д.

Все, что нужно выполнить более одного раза, нуждается в автоматизации. При этом все время, потраченное на автоматизацию, в будущем с лихвой окупится.

8.2. Когда автоматизировать?

С точки зрения того, когда автоматизировать, у вас есть три возможности:

- **при кодировании,**
- **при сборке,**
- **при развертывании.**

Все эти три ключевые фазы разработки программного обеспечения предъявляют свои требования и обладают своими особенностями, даже если вы используете одни и те же инструменты и в принципе делаете то же самое в рамках разных фаз разработки. Например, процесс развертывания кода для его тестирования при разработке может отличаться от развертывания кода «в производство» (в боевых условиях). Два процесса, по сути, должны быть одним и тем же, но в действительности они отличаются.

Развертывание кода на тестовой виртуальной машине отличается от его развертывания на множестве серверов, распределенных по всему миру. Аналогично, модульное тестирование и создание отчетов покрытия кода в вашей локальной среде отличается от модульного тестирования во время процесса сборки, к сожалению. При этом вы, конечно же, должны использовать одинаковые инструменты в обоих случаях.

Мы не будем глубоко погружаться в вопросы развертывания, поскольку во-первых, не это является темой книги, а во-вторых, существующие инструменты различаются в зависимости от масштаба организации и масштаба проектов. Одно дело развернуть программу на один сервер, другое дело — на пятьдесят распределенных по всему миру серверов. Вместо этого мы сфокусируемся на автоматизации разработки и автоматизации сборки, ключевые принципы которых независимы от размера приложения.

8.3. Как автоматизировать?

Автоматизация требует использования каких-то инструментов. Какие инструменты вы будете использовать — собственной разра-

ботки или уже готовые – дело ваше. Главное — конечный результат, который подразумевает создание и тестирование вашего кода простыми и повторимыми способами.

Комплекс мер и инструментов по автоматизации помимо всего прочего должен представлять систему непрерывной интеграции. Непрерывная интеграция (англ. Continuous Integration) — это практика разработки программного обеспечения, которая заключается в выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем.

В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоёмкость интеграции и сделать ее более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий. Непрерывная интеграция является одним из основных приёмов экстремального программирования.

Необходимо также отдавать себе отчет, что помимо автоматизации всего и вся должна сохраняться возможность ручного выполнения тех или иных действий в тех одноразовых исключительных ситуациях, которые, тем не менее, всегда происходят. И разработчикам должно быть понятно, как это сделать.

8.3.1. Автоматизация с непрерывной интеграцией

Одним из самых больших преимуществ полностью автоматизированной среды разработки состоит в возможности непрерывной интеграции. При постоянных сборке и тестировании вы очень быстро фиксируете ошибки — прежде, чем код будет где-либо развернут.

Разработчики отвечают за модульное тестирование перед загрузкой их кода, но модульное тестирование — это только одно из первичных видов тестирования. В системе же непрерывной интеграции код тестируется постоянно и на любые несоответствия.

Так, в рамках автоматизированной системы непрерывной интеграции вы можете произвести параллельное модульное тестирование, сопровождаемое быстрым развертыванием в тестовой среде для некоторого основного интеграционного тестирования. При этом можно запускать не все тесты: обычно вам нужно просто запустить модульные тесты и несколько интеграционных тестов, которые тестируют требуемую функциональность приложения.

В случае выявления проблемы (ошибки) у вас должна быть возможность отследить источник проблемы и выйти на разработчика, ответственного за соответствующий фрагмент кода, а значит и отвечающего за возникший отказ. Далее должны быть приостановлены все сборки, пока проблема не будет решена (а фотография разработчика проблемного кода не будет размещена на «стене позора»). Все внимание теперь быть сосредоточено на исправлении сборки так, чтобы остальная часть команды могла продолжать продвигаться.

В случае успеха сборка может быть в полном объеме развернута в тестовом окружении для более обширного тестирования, после чего она перейдет в другие окружения и в конечном счете будет развернута в производственном окружении, то есть будет предоставлена конечному потребителю (пользователю).

Примечание.

Сайт Integrate Button (http://www.integratebutton.com/), поддерживаемый авторами «Continuous Integration: Improving Software Quality and Reducing Risk», предоставляет много подробностей о процессе непрерывной интеграции.

8.3.2. Автоматизация разработки

Этап непосредственного кодирования — самая неавтоматизированная, самая подверженная ошибкам и одна из наиболее длительных частей любого проекта по разработке программного обеспечения. Пока еще компьютеры не умеют писать код за нас, а мы по-прежнему вводим символы вручную в окне редактора. «Наука» и «искусство» написания программного обеспечения никак не изменилась за последние 40 лет. Именно в начале 1970-ых был представлен первый терминал и произошел настоящий прорыв в разработке программного обеспечения, заключавшийся в переходе с перфокарт на терминальный ввод. Вместо написания программ с по-

мощью перфокарт теперь программы вводятся с клавиатуры непосредственно в компьютер. Но с тех пор ничего не поменялось.

Поэтому любой автоматизированный инструмент, который может хоть что-то упростить, ускорить и автоматизировать задачу создания кода, будет очень кстати. Давайте рассмотрим некоторые из них, которые способны автоматизировать разработку программного обеспечения в части создания кода.

Редактор кода

Самый основной и самый спорный инструмент в арсенале разработчика – это редактор кода.

> **Лучший редактор кода — это тот, который вам больше всего нравится. У каждого редактора есть свои преимущества и недостатки.**

Например, есть необычные IDE с автозавершением ключевых слов и имен переменных, но у них нет такого богатого ряда плагинов, как у некоторых основных редакторов. Используйте тот редактор кода, которым вы довольны. Однако, независимо от того, какой редактор вы используете, убедитесь, что вы ознакомились со всеми его функциями, чтобы максимально упростить вашу работу. Автоматический отступ, автозавершения кода, соответствие фигурных скобок и другие удобные функции существенно упростят вам жизнь. Кроме того, у каждого редактора есть свой собственный набор приемов, о которых вы не знаете и которые наверняка вам пригодятся. Составьте себе за труд поподробнее ознакомиться со своим инструментом.

Что касается моих личных рекомендаций, то я бы порекомендовал редактор VIM (http://www.vim.org/), по крайней мере, для начала.

Модульное тестирование по ходу разработки кода

Имея на руках модульные тесты очень желательно иметь возможность их автоматического выполнения по ходу разработки. Несколько платформ модульного тестирования обеспечивают подходящие для этого функции: возможность выполнения модульного

тестирования из командной строки, динамическое создание отчетов покрытия кода, поддержка нескольких браузеров и т.д. Лично я рекомендую **JUTE** (https://github.com/zzo/JUTE) — удобный инструмент, который делает все вышеперечисленное и даже немного больше. В рамках данного раздела мы сфокусируемся на интеграции JUTE в ваше окружение разработки для максимально безболезненного выполнения модульного тестирования и создания отчетов покрытия кода.

Примечание.

Автор книги – автор JUTE.

JUTE доступен в виде npm-пакета. Самые основные требования JUTE – это требование к использованию YUI3 Test и требования к вашей структуре каталогов. В рамках работы над проектом Yahoo! Mail мои коллеги и я в свое время решили, что самая здравая структура каталогов между кодом приложения и тестовыми сценариями – это зеркально отраженная иерархия. При этом в корневом каталоге создаются два основных подкаталога — `src` и `test`, название которых говорит само за себя. Дерево каталогов `test` зеркально отражает дерево каталогов `src`. Каждый модульный тест из каталога `test` соответствует определенному файлу в каталоге `src`. Конечно, вы можете использовать и другую структуру каталогов, но мой опыт показывает, что такая структура каталогов – наиболее оптимальна.

Начните с установки JUTE:

```
% npm install jute
```

Для его настройки используйте переменные `npm`. Наиболее важной для нас в данном случае является переменная конфигурации — `docRoot`, задающая корневой каталог вашего проекта. По умолчанию, именно в `docRoot` JUTE ожидает увидеть тестовый каталог и каталог с исходниками. Установка каталога `docRoot` выглядит следующим образом :

```
% npm config set jute:docRoot /path/to/project/root
```

Теперь нужно перезагрузить JUTE (вы должны это делать всегда после внесения любых изменений в конфигурацию):

```
% npm restart jute
```

Интерфейс пользователя JUTE можно загрузить, перейдя по следующему адресу в браузере: http://localhost:5883, в результате вы должны увидеть что-то подобное изображенному на рис. 8.1.

Рис. 8.1. Интерфейс пользователя JUTE

На панели **Status** отображаются все "захваченные" браузеры. В данном случае есть только один браузер, который я использую для просмотра этой страницы. Чтобы "захватить" еще какой-то браузер, просто откройте в нем корневую страницу интерфейса пользователя JUTE (http://localhost:5883). При нескольких «захваченных» браузерах все тесты будут запущены параллельно по всем захваченным браузерам.

В центральной области, пока никакие тесты не запущены, просто перечисляются все доступные тесты. Данный список заполняется рекурсивным поиском всех *.html файлов, выступающих в качестве связующих звеньев между кодом и тестом (данные HTML-файлы мы обсуждали в главе 4.), в подкаталоге `test`.

Чтобы запустить любой из тестов, щелкните напротив него по ссылке **Run**. Вы можете выбрать запуск теста с определением покрытия кода или без такового. Для запуска нескольких тестов одновременно установите напротив них флажки, а затем нажмите кнопку **Run Tests**.

Панель **Results** отображает результаты и отчеты о покрытии кода для всех тестов, которые были запущены.

При необходимости вы можете запустить тесты в JUTE из командной строки:

- один тест:

```
% jute_submit_test --test path/to/test.html
```

- несколько тестов:

```
% jute_submit_test --test test1.html --test test2.html
```

Также из командной сроки вы можете прочитать список тестов из `stdin`:

```
% find . -name '*.html' -print | jute_submit_test -test -
```

По умолчанию выходные файлы помещаются в подкаталог `output` каталога `docRoot`. Вы можете изменить этот каталог, используя переменную конфигурации `outputDir`:

```
% jute config set jute:outputDir results
```

Обратите внимание, что полный путь к этому каталогу будет `docRoot + outputDir`, то есть выходной каталог должен находиться в пределах каталога `docRoot`. Все XML-результаты тестов и LCOV-данные будут помещены в этот каталог. Имена файлов с результатами тестов содержат строки **User Agent**, поэтому легко понять, в каком браузере проводился тот или иной тест. Рис. 8.2 показывает внешний вид панели **Results** после запуска теста.

У теста "toolbar" есть результаты в файле формата XML JUnit (см. гл. 4), а файл с выходной информацией содержит отладочную информацию (главным образом вывод YUI Test) и динамически сге-

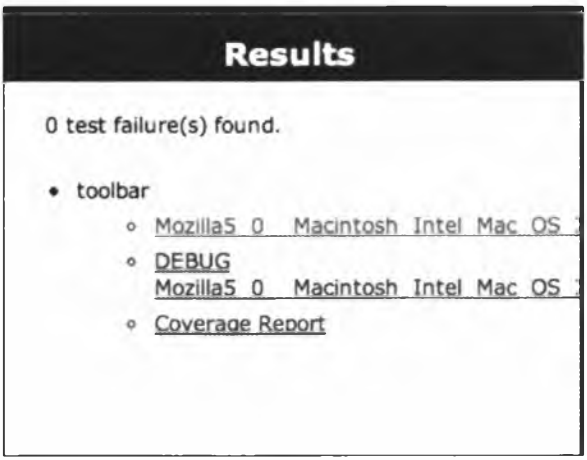


Рис. 8.2. Панель Results после запуска теста в JUTE

нерированный отчет покрытия. Как уже упоминалось, подключить другой браузер к JUTE можно путем посещения страницы JUTE из этого другого браузера. Тогда панель **Status** отобразит оба браузера (рис. 8.3). Теперь к JUTE подключены браузеры Chrome и Firefox.

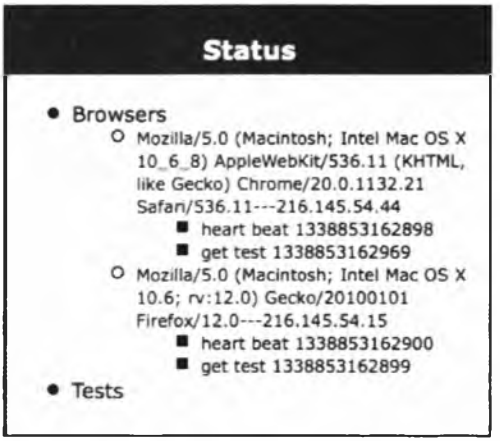


Рис. 8.3. Панель Status

Запуск `jute_submit_tests` приведет к запуску всех тестов параллельно в обоих браузерах, а на панели **Results** будут отображены результаты для Chrome и для Firefox. Рассмотрим содержимое каталога после запуска теста «toolbar» с двумя подключенными браузерами:

```
% ls output/toolbar/
cover.json
lcov.info
lcov-report/
Mozilla5_0__Macintosh_Intel_Mac_OS_X_10_6_8__AppleWebKit536_11__
KHTML__like_Gecko__Chrome20_0_1132_21_Safari536_11-test.xml
Mozilla5_0__Macintosh_Intel_Mac_OS_X_10_6_8__AppleWebKit536_11__
KHTML__like_Gecko__Chrome20_0_1132_21_Safari536_11.txt
Mozilla5_0__Macintosh_Intel_Mac_OS_X_10_6_rv_12_0__Gecko20100101_
Firefox12_0-test.xml
Mozilla5_0__Macintosh_Intel_Mac_OS_X_10_6_rv_12_0__Gecko20100101_
Firefox12_0.txt
```

Обратите внимание, что у нас есть только один отчет о покрытии кода, независимо от того, сколько браузеров подключено. Однако, у нас есть два JUnit XML-файла и два вывода отладчика: по одно-

му набору для каждого браузера (см. строку User String в названии файлов).

Ваш код находится на бездисплейном сервере? Вы нуждаетесь в автоматизации? Вместо использования "реального" браузера JUTE может использовать PhantomJS. Последние версии PhantomJS для запуска не требуют X-сервера. Поэтому интегрировать PhantomJS с JUTE очень просто:

```
% jute_submit_test --test path/to/test.xml --phantomjs
```

Теперь наши результаты будут содержать другой набор результатов JUnit XML и журнал отладки для браузера PhantomJS. В качестве специального бонуса вы также получите снимок экрана "браузера" после выполнения всех тестов (см. рис. 8.4).

Рис. 8.4 показывает, как выглядит бездисплейный браузер PhantomJS на бездисплейной системе Linux. Вы можете увидеть работающие в данный момент тесты в средней панели. Вы также увидите браузер PhantomJS в панели **Status**. Выполняемые в данный момент тесты подсвечиваются желтым в панели **Status**.

JUTE не требует внесения каких-либо изменений в ваши тесты (или в тестируемый код) для получения динамических отчетов покрытия кода. Поэтому при желании вы все еще можете вручную загрузить ваши связующие HTML-файлы непосредственно в браузер и выполнить их, если это необходимо.

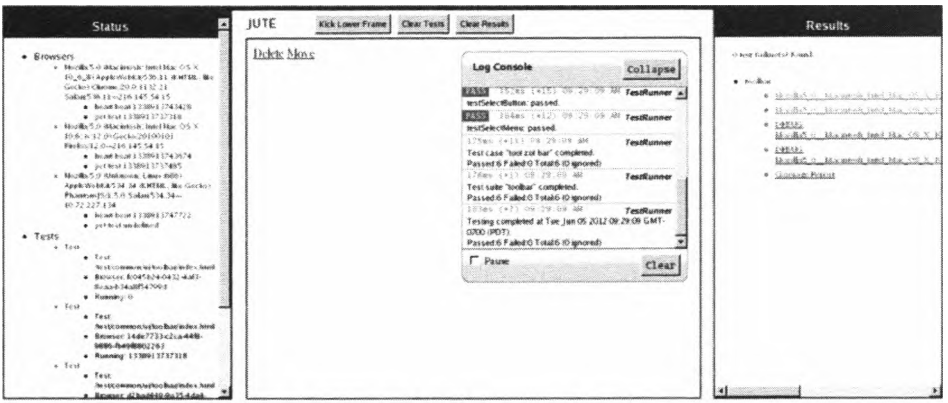


Рис. 8.4. JUTE выполняет тесты

С помощью динамической генерации отчета покрытия кода JUTE позволяет вам легко отслеживать ход тестирования, независимо от того, как вы тестируете код — через командную строку или интерфейс пользователя. Это большой шаг к совершенному JavaScript.

Просмотр кода. Как автоматизировать?

Про полезность просмотра и ревью кода вашими коллегами было много сказано в начале книги. Совместные просмотры кода помогают придерживаться единого стиля кодирования и помогают новичкам более быстро ознакомиться с кодом. И это уже не говоря о том, что члены команды в ходе такого просмотра делятся своими знаниями, что есть очень хорошо! Однако, просмотр гигантского блока JavaScript-кода – это занятие на самое увлекательное и благодарное. Хорошо бы для этого использовать какие-то инструменты. Можно использовать простое решение — общение по электронной почте. Но лучше использовать более специализированные средства. Мне нравится приложение **Review Board (http://www.reviewboard.org/)** – пакет **RBTools**. Данное приложение относительно просто установить и оно имеет хорошую интеграцию с CVS, Subversion, Git (и другими системами управления версиями исходного кода) и баг-трекером Bugzilla.

После установки пакета RBTools вам станет доступен сценарий командной строки `post-review`, который используется для создания запросов на просмотр кода. У **Review Board** также есть HTTP JSON API (**http://www.reviewboard.org**), который вы можете использовать для более сложной интеграции.

В общем, для просмотров кода я настоятельно рекомендую использовать **Review Board** или подобные приложения.

Ловушки предварительной фиксации (коммитов)

Большинство репозитариев управления исходным кодом предоставляют ловушки предварительной фиксации (pre-commit hooks). Это означает, что вы можете осуществить произвольные проверки перед загрузкой кода и отмену фиксации (коммита) в случае необходимости. Проверьте, что эта фиксация не ошибочна, убедитесь, что было проведено модульное и дымовое (smoke)[1] тестирование,

1 Smoke Test (англ. Smoke testing, дымовое тестирование) в тестировании программ-

проверьте код с помощью JSLint и принудительно выполните любой вид анализа и проверки перед тем, как код попадет в производство.

Способы установки ловушек предварительной фиксации (коммита) различны для каждого репозитария. Но как правло это обычные сценарии оболочки, которые возвращают отличное от 0 значение, чтобы прервать фиксацию. Конечно, не все разработчики репозитариев следуют этому правилу, поэтому тут лучше использовать принцип "доверяй, но проверяй" и удостовериться, как у вас есть на самом деле.

Рассмотрим Perl-сценарий ловушки предварительной фиксации для Subversion (SVN), который запустит JSLInt на каждом файле и возвратит общее число ошибок (если они будут найдены в файлах), и если это значение будет отлично от 0, то фиксация (коммит) будет отменена:

```
#!/usr/bin/perl
# Передано от SVN
my $REPO = shift;
my $TXN = shift;
# Устанавливаем PATH
$ENV{PATH} = "$ENV{PATH}:/usr/bin:/usr/local/bin";
# Протоколируем сообщения
my @logmsg = `svnlook log -t "$TXN" "$REPO"`;
#print STDERR @logmsg;
# Получаем файл с изменениями
my @changes = `svnlook changed --transaction "$TXN" "$REPO"`;
my $failed = 0;
#print STDERR @changes;
# Находим JS-файлы
foreach (@changes) {
   my($cmd, $file) = split(/\s+/);
   # Только *.js файлы, которые не помечены как удаленные
   if ($file =~ /\.js$/ && $cmd ne 'D') {
      # Текст измененного файла:
      # my @cat = `svnlook cat "$REPO" --transaction "$TXN" $file`;
```

ного обеспечения означает минимальный набор тестов на явные ошибки. «Дымовой тест» обычно выполняется самим программистом; не проходящую этот тест программу не имеет смысла отдавать на более глубокое тестирование. Первое свое применение этот термин получил у печников, которые, собрав печь, закрывали все заглушки, затапливали её и смотрели, чтобы дым шёл только из положенных мест.

```
        # ИЛИ просто получаем непосредственно файл с предв. изменениями
          my @jslint = `/usr/local/bin/jslint $file`;
        if ($?) {
          print STDERR '-' x 20, "\n";
          print STDERR "Ошибки JSLint в $file:\n";
          print STDERR '-' x 20;
          print STDERR @jslint;
          $failed++;
        }
    }
}
# STDERR возвращается к клиенту в случае отказа
exit $failed;
```

Этот пример получает сообщения протоколирования, но не использует их:

```
my @changes = `svnlook changed --transaction "$TXN" "$REPO"`;
```

А этот код также демонстрирует, как непосредственно получить текст измененного файла:

```
# my @cat = `svnlook cat "$REPO" --transaction "$TXN" $file`;
```

Захват текста сообщения о фиксации позволяет вам предписывать определенный формат для самого сообщения, гарантируя, что ID ошибки, связанной с этой фиксацией будет дополнено допустимыми именами пользователей. Другими словами, каждая фиксация (коммит) исправляет какую-то ошибку (баг) в коде. Обычно с фиксацией связывается только ID ошибки, но благодаря возможности захвата текста о фиксации вы можете получить имя пользователя, который произвел фиксацию.

Ловушки предварительной фиксации Git работают аналогично — в случае сбоя фиксации возвращается ненулевое значение. Разница заключается лишь в том, как захватываются сообщения протоколирования.

Давайте создадим предварительную фиксацию Git, которая отклонит любую фиксацию в случае любого из следующих нарушений:

длина функции больше 30 строк, у функции более пяти аргументов, цикломатическая сложность больше 9.

Мы будем использовать npm-пакет **jscheckstyle (https://github.com/nomiddlename/jscheckstyle)** для получения значения сложности для каждой функции:

```
% npm install jscheckstyle -g
```

Этот пакет добавляет преимущества вывода в Jankins-совместимом формате. Рассмотрим сценарий предварительной фиксации (/git/hooks/pre-commit):

```
#!/usr/bin/perl
# Get file list
my @files = `git diff --cached --name-only HEAD`;
my $failed = 0;
foreach (@files) {
   # Проверяем только файлы *.js
   if (/\.js$/) {
      my @style = `jscheckstyle --violations $_`;
      if ($?) {
         # Отправляем сообщение об ошибке клиенту
         print @style;
         $failed++;
      }
   }
}
exit $failed;
```

Этот код просто получает список файлов, которые должны быть зафиксированы и запускает jscheckstyle для каждого из них. Если любой из файлов провалит проверку, клиенту будет отправлено сообщение об ошибке и фиксация будет отменена.

Фиксация сложного файла из моего npm-пакета **Injectify**:

```
% git commit index.js
The "sys" module is now called "util". It should have a similar
interface.
jscheckstyle results — index.js
```

Line	Function	Length	Args	Complex..
30	Injectify.prototype.parse	197	2	27
228	Injectify.prototype.statement	42	1	10

Функции `parse` и `statement` очень сложны. И поэтому фиксация была прервана из-за превышения установленных нами ранее граничных значений сложности. Однако, при задании ограничений, вы можете использовать свои значения и свои переменные конфигурации Git для определения пороговых значений для вашего окружения и вычисления сложности.

А теперь представьте, что весь код автоматически проходит подобную проверку, верификацию и тестирование до помещения в репозитарий. Тогда любой код, переданный в репозитарий, должен считаться готовым к работе (запуску в производство («продакшн»).

Другие инструменты разработчика

Результаты работы JSLInt на некоторых файлах может свети с ума из-за количества выявленных ошибок. Автоматически исправить базовые ошибки, выявленные JSLInt, можно с использованием пакета **fixmyjs**. Он автоматически исправляет основные ошибки, найденные JSLInt, таким образом избавляя вас от огромного количества работы и уберегая вашу нежную психику. Использовать пакет просто:

```
% npm install fixmyjs -g

% fixmyjs myFile.js
```

В предыдущем коде файл *myFile.js* будет обновлен, вам же будет показана разница между его новой версией и более ранней. Если вы

опасаетесь, что будут внесены неправильные изменения, сделайте тестовый прогон, чтобы увидеть, что сделает программа:

```
% fixmyjs -r myFile.js
```

Или даже создайте патч-файл вместо изменения исходного файла:

```
% fixmyjs -p myFile.js
```

Этот превосходный инструмент позволит начать новую жизнь с вашим JSLint. Вы будете приятно удивлены как тому, что он находит, так и тому, что он в состоянии не только найти, но и исправить ошибки. Как только ваш код станет JSLint-дружественным, можете полагаться на этот инструмент (JSLint + fixmyjs) в части автоматического исправления ошибок и поддержке вашего кода «в чистоте».

8.3.3. Автоматизация среды сборки

Автоматизация, произведенная в среде сборки, является всего надмножеством того, что происходит в среде разработки. Вы же не хотите, чтобы произошел сбой сборки по причине высокой цикломатической сложности, которая не была проверена в среде разработки? Или вы же не хотите, чтобы произошел сбой из-за ошибок JSLint, которые не были выявлены и устранены в среде разработки?

С другой стороны разработчики ненавидят сюрпризы в среде сборки, поэтому на данном этапе не нужно вносить в код никакие изменения, которых не было в среде разработки. Короче говоря, изменения в коде нужно вносить только в среде разработки.

Сборка

Для осуществления сборки необходима какая-либо система сборки — например, makefile, Ant XML-файл или что-то еще. Как и ваша приложение, сборка не должна быть монолитным монстром, она должна состоять из ряда мелких частей. Каждая часть должна реализовывать какое-то одно назначение, и цель сборки — собрать все эти части в определенном порядке.

Нам не нужно, чтобы в процессе сборки минификация и сжатие (оптимизация кода через подмену длинных имен короткими) происходили за одно действие. Это два различных действия. Сборка должна быть параметризована — через командную строку или любой другой настраиваемый источник. Разработчикам наверняка захочется запустить части вашей сборки в своих окружениях или же запустить полную сборку для разворачивания программы в их тестовых окружениях в то время как «реальные» сборки будут работать на выделенной машине в производственном (боевом) режиме. Таким образом, сборка должна быть достаточно гибка, чтобы позволять делать с собой разные такие штуки.

Идеально, чтобы разработчики и лица, ответственные за сборку, могли использовать одни и те же сценарии, чтобы произвести аналогичную или полностью идентичную сборку независимо от окружения и настроек.

Далее мы рассмотрим, как можно автоматизировать процесс сборки.

Jenkins

Jenkins (http://jenkins-ci.org/), ранее Hudson, отличный инструмент для автоматизации процесса сборки. Jenkins обладает множеством различных плагинов, огромным числом настроек и удобным интерфейсом пользователя, что выгодно выделяет его среди других подобных инструментов.

Установка Jenkins довольно проста, особенно если у вас есть опыт развертывания WAR (Web-архивов) файлов — именно в этом формате упакован Jenkins.

Загрузите последнюю версию (или версию с длительным периодом поддержки — Long Term Support, LTS), разверните WAR-файл в надлежащий каталог, перезапустите ваш сервер, и Jenkins будет готов к работе.

Если вы хотите ускорить сей процесс, просто загрузите WAR и запустите его:

```
% java -jar jenkins.war
```

Jenkins работает на 8080 порту. На рис. 8.5 изображена домашняя административная страница Jenkins.

Рис. 8.5. Домашняя административная страница Jenkins

Прежде, чем мы создадим наше первое задание, давайте поговорим о некоторых плагинах, которые могут быть вам интересны. Щелкните по ссылке **Manage Jenkins**, и вы увидите множество параметров конфигурации. Нас в данный момент интересуют плагины, поэтому щелкните по ссылке **Manage Plugins**. В менеджере плагинов увидите все установленные на данный момент плагины и сможете при желании установить новые. Раньше по умолчанию были доступны плагины, обеспечивающие поддержку CVS и Subversion, а при желании можно было установить поддержку Git. Теперь же по умолчанию ни один из плагинов не устанавливается, и необходимые плагины нужно устанавливать самостоятельно. Например, если вы нуждаетесь в поддержке Git, щелкните по вкладке **Available** и найдите плагин **Git Client** (рис. 8.6).

Выберите флажок **Git Client Plugin** и нажмите кнопку **Install without restart** для установки Git-плагина без перезагрузки, что полезно, если вы уже добавили много заданий Jenkins и перезагружать

Рис. 8.6. Получение плагина Git

его нельзя. Также можно нажать кнопку **Download now and install after restart** — в этом случае плагин будет загружен сейчас, а установлен только после перезагрузки Jenkins.

Создание проекта Jenkins

Теперь вы можете создать проект. При этом вам нужно указать Jenkins две вещи: где найти репозитарий с вашим исходным кодом и как его собрать. Jenkins проверит ваш код и запустит вашу команду сборки. Если команда сборки вернет 0, то это означает, что создание вашей сборки прошло успешно.

Итак, перейдите на главную страницу Jenkins и щелкните по ссылке **New Job**. Дайте вашему заданию имя. Внимание! Не используйте пробелы и пробельные символы в имени проекта! Затем щелкните мышкой по **Build a free-style software project**, нажмите кнопку **OK** и перед вами все будет готово для конфигурирования нового проекта. На рис. 8.7 изображена домашняя страница проекта Jenkins.

Рис. 8.7. Домашняя страница проекта Jenkins

Щелкните по ссылке **Configure** и заполните базовые параметры проекта. Для Subversion-проекта необходимо указать URL репозитария, как показано на рис. 8.8. Для локального репозитария Subversion, если вы установили плагин Git, вы можете указать Git URL.

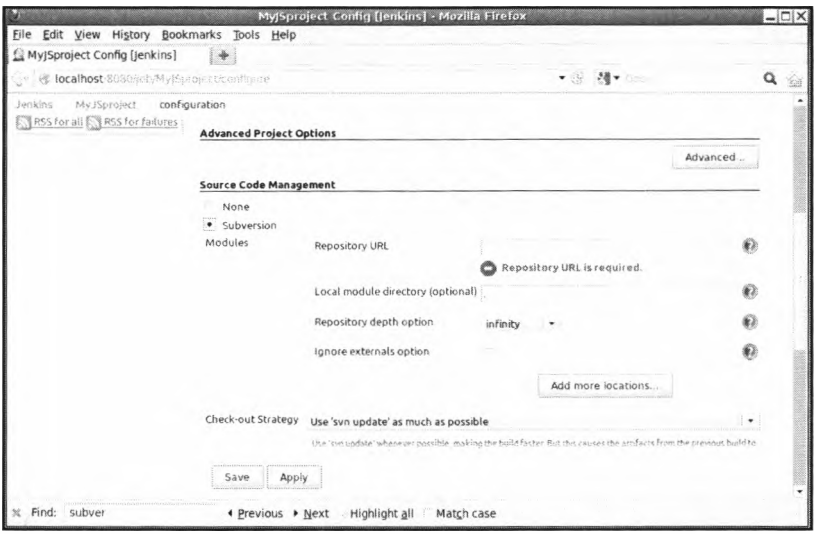


Рис. 8.8. Настройка SVN-репозитария в Jenkins

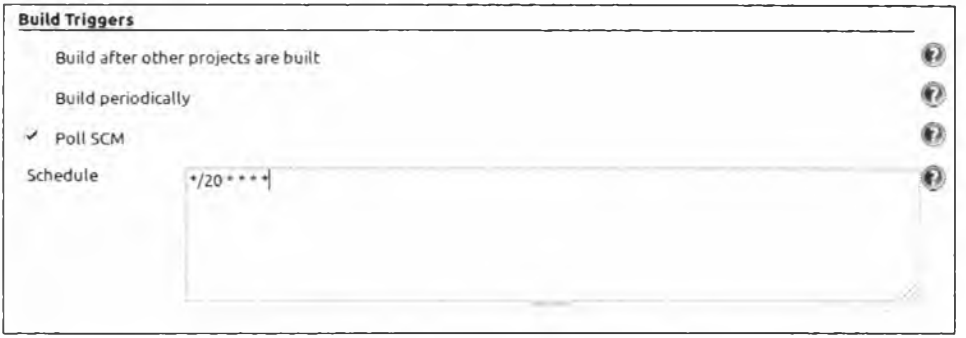


Рис. 8.9. Опрос Subversion

В разделе **Build Triggers** вы можете включить триггер **Poll SCM**, позволяющий неоднократно проверять любые изменения. На рис. 8.9 показано, что я опрашиваю Subversion каждые 20 минут. Если что-то изменилось, будет начата сборка (вот она наша непрерывная интеграция!)

Наконец, вы должны указать Jenkins, что нужно сделать для сборки репозитория (например, указать команду сборки) и что сделать после сборки.

Рассмотрим Makefile, который произведет модульное тестирование всего кода и сожмет код, разместив его в каталоге *release*, который может быть впоследствии размещен на веб-сервере, если все нормально:

```
DO_COVERAGE=1
RELEASE=release
UGLIFY=uglifyjs
SRC := $(shell find src -name '*.js')
OBJS := $(patsubst %.js,%.jc,$(SRC))
%.jc : %.js
        -mkdir -p $(RELEASE)/$(@D)
        $(UGLIFY) -o $(RELEASE)/$(*D)/$(*F).js $<
prod: unit_tests $(OBJS)
setupJUTE:
ifdef WORKSPACE
        npm config set jute:docRoot '$(WORKSPACE)'
        npm restart jute
endif
```

```
server_side_unit_tests: setupJUTE
        cd test && find server_side -name '*.js' -exec echo
{}?do_coverage=$(DO_COVERAGE)" \; | jute_submit_test --v8 --test -
client_side_unit_tests: setupJUTE
        cd test && find client_side -name '*.html' -exec echo
"{}?do_coverage=$(DO_COVERAGE)" \; | jute_submit_test --v8 --test -
unit_tests: server_side_unit_tests client_side_unit_tests
.PHONY: server_side_unit_tests client_side_unit_tests unit_tests
setupJUTE
```

Этот Makefile использует UglifyJS (**https://github.com/mishoo/UglifyJS**) для сжатия/оптимизации кода, но вы также можете использовать и другие программы, например YUI Compressor (**https://github.com/yui/yuicompressor/issues**) и Google Closure Compiler (**https://developers.google.com/closure/compiler/**). Файл Makefile использует каталоги src и test для исходного кода и тестовых сценариев, соответственно.

Выполнение цели `prod` (используется по умолчанию) произведет модульное тестирование всего кода и минимизирует весь код, если все тесты будут успешно пройдены. Тесты запускаются через V8, следовательно, вы не нуждаетесь в браузере. Однако, это может не очень подходить для тестирования всего клиентского JavaScript, поэтому вы можете использовать PhantomJS (`-phantomjs`) или Selenium (`-sel_host`) для выполнения тестов через реальный браузер.

Вы можете даже использовать «захваченные» браузеры, по крайней мере, один захваченный экземпляром JUTE браузер, запущенный на машине. Обратите внимание, в этом нашем случае мы должны настроить JUTE так, чтобы при запуске Jenkins JUTE "знал", где находится корневой каталог документов, пока Jenkins собирает проект. Если у вас есть выделенное окружение для сборки, вы можете настроить JUTE всегда использовать рабочее пространство Jenkins. Динамическую информацию о покрытии кода можно добавить в качестве параметров команды `make`:

```
% make prod DO_COVERAGE=0
```

Здесь проявляется некоторое волшебство make-файла. Все ваши JS-файлы будут преобразованы в их сжатые версии в дереве каталогов, которое зеркально отражает дерево *src* в каталог release. Как только

Рис. 8.10. Настройка команды сборки в Jenkins

сборка будет закончена, вам нужно просто создать программой *tar* архив каталога release и поместить его на ваши веб-серверы.

Итак, давайте вернемся в Jenkins. Просто добавьте команду `make` в качестве команды сборки, как показано на рис. 8.10.

Сохраните проект и запустите сборку, нажав **Build Now**. Если все будет хорошо, через некоторое время вы получите сборку.

Модульное тестирование средствами Selenium

Теперь давайте вернемся к Selenium. Здесь запуск тестов во время сборки происходит в "реальных" браузерах. Вы можете хранить захваченные браузеры в экземпляре JUTE, запущенном в среде сборки, но есть более быстрый способ, особенно когда при увеличении числа модульных тестов, время, необходимое на их запуск и выполнение вашей сборки, также увеличивается.

К счастью, при использовании сетки Selenium JUTE может параллелизировать модульное тестирование по всем узлам Selenium с использованием единственного концентратора сетки. Также вы можете сгенерировать снимки экрана в конце каждого теста. Снимки экрана сохраняются вместе с результатами тестов для простой отладки отказов тестов, если вывода журнала недостаточно.

Настроить сетку Selenium просто. Идея заключается в использовании одного концентратора, к которому будут подключаться узлы Selenium. Любой внешний процесс, который хочет управлять брау-

зерами Selenium, просто подключается к этому концентратору и передает ему запросы. Результаты передаются обратно от концентратора запрашивающей стороне.

Теперь перейдите на сайт SeleniumHQ (**http://docs.seleniumhq.org/download/**) и получите последнюю версию Selenium-сервера. Этот единственный JAR-файл (`selenium-server-standalone-3.141.59.jar`) инкапсулирует и концентратор, и узел, необходимые для построения вашей сетки. На одном компьютере можно запустить несколько узлов Selenium. Фактически, на одном компьютере может работать и концентратор, и узлы Selenium. Однако одновременный запуск множества узлов на одном компьютере может вызвать проблемы с производительностью. Экспериментальным способом я определил, что на одном компьютере можно запускать 2-3 узла Selenium. И не надейтесь, что вы сможете запустить на одном компьютере более 10 экземпляров узлов Selenium. Кроме того, к концентратору могут подключиться узлы, работающие под управлением разных операционных систем. Так, к концентратору под управлением Windows могут подключиться узлы, работающие на Mac, Linux или Windows.

Скопируйте JAR сервера Selenium на компьютер концентратора и компьютеры узлов и запустите его. Запустить концентратор можно так:

```
% java -jar selenium-server-standalone-3.141.59.jar -role hub
```

По умолчанию концентратор прослушивает порт 4444. Вы можете изменить это значение с помощью опции `-port`. Однако, поскольку все узлы и клиенты Selenium используют именно порт 4444, имейте в виду, что вам придется также изменить номер порта для всех этих процессов.

Теперь запустим несколько узлов. Для этого используем следующую команду:

```
% java -jar selenium-server-standalone-3.141.59.jar -role node -hub http://<HUB HOST>:4444/grid/register
```

Узлы Selenium должны знать, где находится концентратор (компьютер и порт), иначе они не смогут подключиться к нему. По умолчанию узлы прослушивают порт 5555. Поэтому если вы запускаете

несколько узлов на одном компьютере, вы должны выбрать разные значения портов для каждого узла, кроме первого. Номера портов, которые вы будете использовать для узлов, не имеют значения, они нужны лишь для открытия порта. Итак, для запуска другого экземпляра узла на том же компьютере, используйте команду (в одну строку):

```
% java -jar selenium-server-standalone-3.141.59.jar -role node -hub http://<HUB
HOST>:4444/grid/register -port 6666
```

Теперь у нас есть два узла, запущенных на одном компьютере. Эти узлы могут быть на одной машине с концентратором или же запущенными на выделенном компьютере даже с другой операционной системой. Вы можете настроить каждый узел, чтобы он сообщал концентратору, какие браузеры и какую операционную систему поддерживает узел. Для этого используйте параметр `-browser` командной строки. Например, следующий параметр сообщает концентратору, что создаваемый узел поддерживает браузер Firefox версии 13 под управлением Linux:

```
-browser browserName=firefox,version=13,platform=LINUX
```

Подробнее о настройке узлов Selenium вы можете прочитать на следующем сайте: **https://code.google.com/p/selenium/wiki/Grid2**.

Теперь JUTE может использовать вашу сетку Selenium для параллельного тестирования. Рассмотрим следующую команду (все в одной строке):

```
% jute_submit_test --sel_host <grid host> --sel_port 4444 --seleniums 5 --test -
```

Здесь `<grid host>` — имя компьютера, на котором работает Selenium-концентратор, а параметр `--sel_port` задает его порт. Параметр `--sel_port` нужно указывать, только если ваш концентратор работает на порту, отличном от 4444. Параметр `-seleniums` говорит JUTE как нужно параллелизировать модульное тестирование. В этом примере все ваши тесты будут разделены на 5 частей. Каждую часть будет выполнять отдельный узел. Скажем, если у вас 1000 тестов, то каждый узел Selenium выполнит 200 тестов. Yahoo! Mail использует эти настройки в полный рост и выполняет около 3000 тестов за 10 минут всего на четырех Selenium-узлах.

JUTE делает снимок экрана браузера после каждого неудачного теста. Если вы хотите создавать снимки экрана после каждого теста, независимо от его результата, используйте опцию `--snapshot`.

Чтобы указать, в каком браузере должны быть выполнены все тесты, используйте опцию `--sel_browser` команды `jute_submit_test`.

Теперь соберем все в месте. Цель вашего make-файла, позволяющая запустить все ваши тесты за один раз через Selenium, может выглядеть так:

```
selenium_tests:
cd test && find . -name '*.html' -printf '%p?do_coverage=1\n' |
jute_submit_test --sel_host 10.3.4.45 --seleniums 5 --test -
```

Далее для запуска всех тестов выполните команду **make** с этой целью:

```
make selenium_tests
```

Вывод модульного теста

Все, что остается сделать — заставить Jenkins распознать вывод модульного теста (файлы в формате JUnit XML, которые генерирует JUTE), данные покрытия (которые также генерирует JUTE) и любые другие данные сборки, которые понадобятся Jenkins.

Удобно, что JUTE помещает весь вывод теста в один настраиваемый каталог, который по умолчанию называется output. Этот каталог находится в *jute:docRoot*, который также является рабочим каталогом Jenkins. Чтобы Jenkins распознал результаты тестов, нужно просто

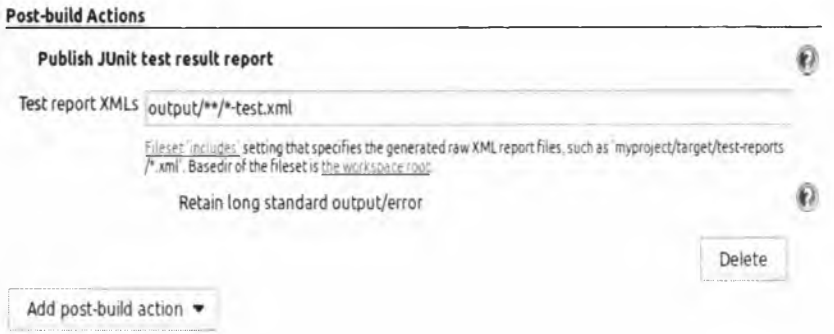


Рис. 8.11. Публикация вывода JUnit XML в Jenkins

настроить действие после сборки. Для этого выберите действие **Publish JUnit test result report** и установите его параметры, как показано на рис. 8.11.

Теперь мы можем начать сборку, щелкнув по ссылке **Build Now**. Jenkins отобразит график тестирования на панели инструментов нашего проекта (рис. 8.12). Также нужно отметить ссылку **Latest Test Result**, позволяющую просмотреть результат последнего теста.

Рис. 8.12. График тестирования

Вывод покрытия кода

Интеграция покрытия кода, к сожалению, более сложна в последней версии Jenkins, поскольку отсутствует поддержка формата LCOV. Так, мы должны сначала агрегировать все отдельные файлы покрытия в один файл и затем преобразовать тот файл в другой формат, поддерживаемый Jenkins.

Агрегация всех отдельных файлов покрытия (файлы `lcov.info` в каждом каталоге) в один-единственный файл осуществляется командой `lcov`. Рассмотрим соответствующее правило для нашего make-файла:

```
OUTPUT_DIR=output
TOTAL_LCOV_FILE=$(OUTPUT_DIR)/lcov.info
make_total_lcov:
   /bin/rm -f /tmp/lcov.info ${TOTAL_LCOV_FILE}
```

```
    find $(OUTPUT_DIR) -name lcov.info -exec echo '-a {}' \;
  | xargs lcov > /tmp/lcov.info
    cp /tmp/lcov.info ${TOTAL_LCOV_FILE}
```

Этот код собирает все сгенерированные файлы lcov.info в каталог output и запускает команду `lcov -a ...` для агрегации всех этих файлов в один файл `lcov.info`, который создается в корне каталога output.

Теперь, когда у нас есть один большой файл lcov.info, нам нужно конвертировать его в формат Cobertura XML. Мы можем сделать это с помощью сценария, предоставленного специальным проектом на GitHub: **https://github.com/eriwen/lcov-to-cobertura-xml**. Цель make-файла будет выглядеть так:

```
cobertura_convert:
    lcov-to-cobertura-xml.py $(TOTAL_LCOV_FILE) -b src
-o $(OUTPUT)/cob.xml
```

Добавление этих двух целей в исходную цель `unit_tests` сгенерирует всю необходимую Jenkins информацию для отображения покрытия кода.

Следующий шаг — установка Jenkins-плагина Cobertura, в результате которой должна стать доступной опция **Publish Cobertura Coverage Report** в списке действий после сборки. Поэтому установите плагин и укажите ему, где будут находиться файлы Cobertura XML (каталоге output), как показано на рис. 8.13.

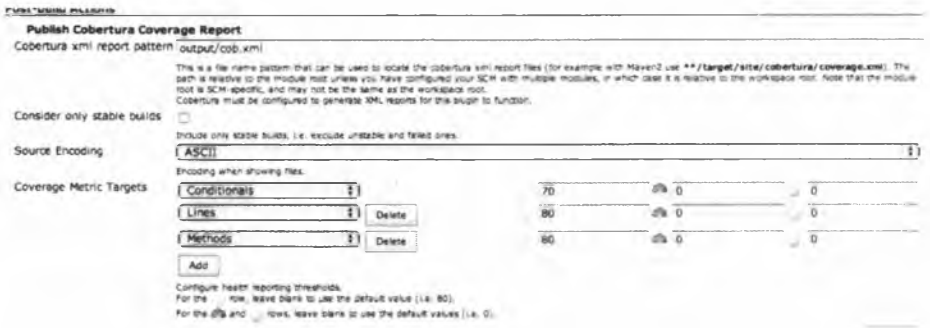


Рис. 8.13. Настройка вывода Cobertura в Jenkins

Рис. 8.14. График покрытия на панели инструментов проекта

Пересоберите ваш проект, и у вас на панели инструментов появится ссылка **Coverage Report**, а также станет доступен график покрытия, что и показано на рис. 8.14.

Щелкните по ссылке **Coverage Report** , чтобы получить более детальную информацию о покрытии кода по отдельным файлам. На рис. 8.15 показано, как выглядит отчет покрытия. Данный вывод очень похож на вывод LCOV, который мы видели в главе 5.

Рис. 8.15. Отчет покрытия

Сложность

Мы добавим еще один полезный плагин Jenkins. Ранее мы обсуждали npm-пакет **jscheckstyle**, который может вывести свой отчет в формате Checkstyle. Вы бы и не знали об этом формате, если бы в Jenkins не было специального плагина, поддерживающего данный формат. Объединив **jscheckstyle** и Jenkins, вы сможете видеть информацию о сложности вашего проекта в панели инструментов Jenkins, что очень удобно.

Процесс стандартный: нужно установить плагин Jenkins, обновить make-файл для генерации требуемых файлов, а затем настроить ваш проект на использование установленного плагина и в настройках плагина указать каталог, в котором будут находиться сгенерированные файлы. Вроде бы все ясно, так давайте приступим к настройке!

Первым делом установите Jenkins-плагин **Checkstyle**, а затем добавьте следующие цели в ваш make-файл, чтобы генерировать информацию о сложности с помощью **jscheckstyle**:

```
CHECKSTYLE=checkstyle
STYLE := $(patsubst %.js,%.xml,$(SRC))
%.xml : %.js
  -mkdir -p $(CHECKSTYLE)/$(@D)
  -jscheckstyle --checkstyle $< > $(CHECKSTYLE)/$(*D)/$(*F).xml
```

Все очень подобно коду для UglifyJS, приведенному ранее. Мы преобразуем JavaScript-файлы из каталога `src` в XML-файлы в каталоге, зеркально отображенном Checkstyle. Обратите внимание, что **jscheckstyle** завершит работу, если найдет какие-либо нарушения, но это остановит сборку! В случае с предварительной фиксацией это было полезно, но останавливать сборку из-за сбоя создания отчета сложности — не самое хорошее решение.

Добавьте `$(STYLE)` к цели `prod`:

```
prod: unit_tests $(OBJS) $(STYLE)
```

После этого все эти файлы будут сгенерированы при следующей сборке. Теперь расскажем об этом Jenkins: на экране **Configure** вашего проекта добавьте действие после сборки **Publish Checkstyle analysis results** и настройте его, как показано на рис. 8.16.

Рис. 8.16. Настройка Checkstyle в проекте Jenkins

Здесь мы указываем плагину, что искать необходимые файлы нужно в подкаталоге `checkstyle` нашего рабочего каталога. Нажмите ссылку **Build Now** и приготовьтесь получить результаты (рис. 8.17).

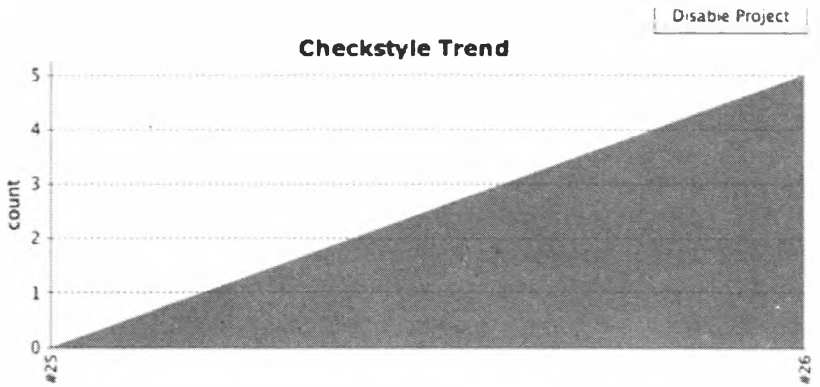


Рис. 8.17. График Checkstyle на домашней странице проекта Jenkins

Данный график появится на домашней странице вашего проекта. Щелчок по графику покажет историю и все текущие предупреждения, если таковые имеются, в том числе и предупреждения в новой сборке.

JSLint + Jenkins

Процесс интеграции JSLint с Jenkins аналогичен предыдущим интеграциям — для этого вам нужен плагин **Violations**. Установите его. Кстати, приятно, что он также обрабатывает Checkstyle-файлы. Но об этом мы поговорим позже, а пока вам нужно знать, что плагин **Violations** требует JSLInt вывода в определенном XML-формате, который легко сгенерировать:

```
<jslint>
  <file name="<full_path_to_file>">
    <issue line="<line #>" reason="<reason>" evidence="<evidence" />
    <issue ... >
  </file>
</jslint>
```

Если вы запустите npm-пакет **jslint** с аргументом `--json`, вы произведете простое преобразование из формата JSLint JSON в этот XML-формат. Моя реализация этого преобразования доступна по адресу: **https://github.com/zzo/TestableJS/blob/master/hudson_jslint.pl**.

Идея, как обычно, заключается в следующем: запустить JSLInt на нашем коде, получить вывод в формате XML и указать плагину **Violations**, где найти этот XML. Первым делом давайте добавим цель в наш make-файл:

```
JSLINT=jslint
JSL := $(patsubst %.js,%.jslint,$(SRC))
%.jslint : %.js
   -mkdir -p $(JSLINT)/$(@D)
   ./hudson_jslint.pl $< > $(JSLINT)/$(*D)/$(*F).jslint
```

Затем нужно добавить цель `JSL` в нашу цель `prod`:

```
prod: unit_tests $(OBJS) $(STYLE) $(JSL)
```

Все готово. Конечно, если нужно запустить JSLint отдельно, просто добавьте эту цель:

```
jslint: $(JSL)
```

Теперь настройте плагин **Violations**, указав расположение XML-файлов ***.jslint** (см. рис. 8.18).

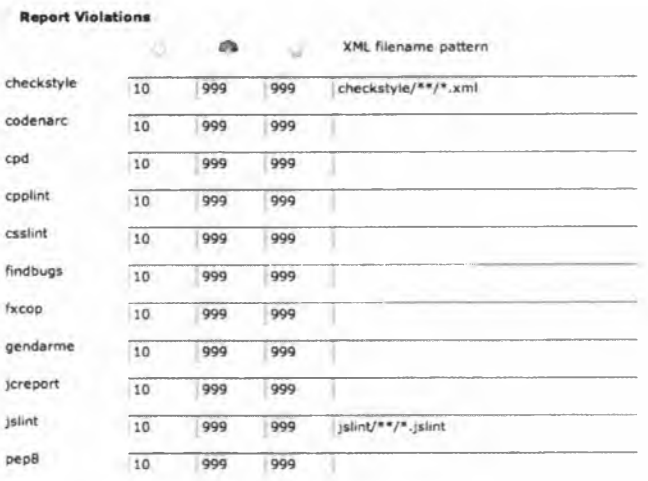


Рис. 8.18. Настройка плагина Violations в Jenkins

Обратите внимание на то, как я настроил размещение вывода **jscheckstyle.**

Теперь заново соберите ваш проект и вы увидите график, показанный на рис. 8.19. Щелчок на графике покажет все ошибки JSLint и Checkstyle в вашем коде. Просматривая ваши файлы в месте нарушения вы увидите значок нарушения. Если подвести к нему указатель мыши, можно увидеть подробную информацию об ошибке.

Рис. 8.19. График нарушений проекта

Дублирующийся код

Выполнение сборки очень полезно с точки зрения нахождения всего дублирующего кода в вашем проекте. Некоторые специальные инструменты анализируют пробелы, комментарии, сравнивают базовые маркеры и дерево синтаксиса. В результате могут быть найдены или двойные, или очень похожие фрагменты кода.

Один из таких инструментов – утилита командной строки dupfind (**https://github.com/sfrancisx/dupfind**), которая послужит отличным дополнением к вашему автоматизированному процессу сборки. Утилита анализирует весь ваш код и пытается найти дублирующиеся секции. При этом она может выводить CPD-совместимый XML-файл (**http://pmd.sourceforge.net/pmd-5.0.4/cpd.html**), а для поддержки файлов в этом формате существует специальный

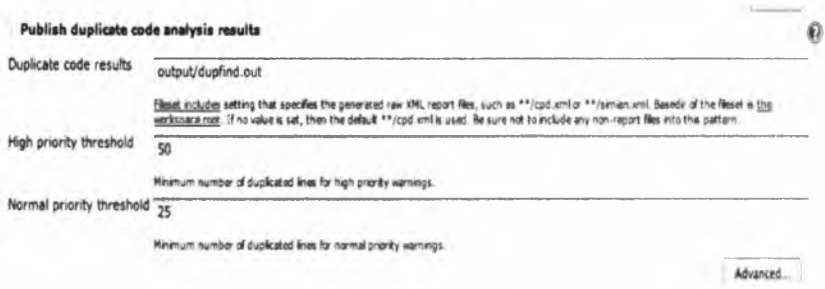


Рис. 8.20. Публикация дублирующегося кода в Jenkins

плагин Jenkins: установите плагин Duplicate Code Scanner (**https://wiki.jenkins-ci.org/display/JENKINS/DRY+Plugin**), он же DRY Plugin, в результате будет создано действие **Publish duplicate code analysis results**, которое вы можете использовать в своем проекте (рис. 8.20).

Пороговые значения приоритета позволяют вам настроить, сколько дублирующих строк является нормальным. Плагин тогда будет отслеживать число дублирующихся строк и выведет график всех дублирующихся секций кода. В расширенных настройках (кнопка **Advanced**) можно установить лимиты, когда плагин пометит сборку нестабильной (unstable) или неудачной (failed), см. рис. 8.21.

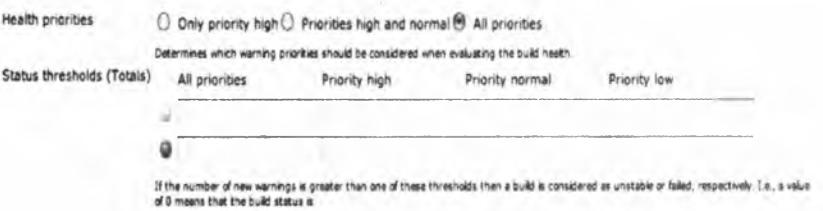


Рис. 8.21. Конфигурация пороговых значений дублирования кода в Jenkins

Установить **dupfind** просто. Скачайте `dupfind.jat` по адресу **https://github.com/sfrancisx/dupfind** и добавьте следующие строки в ваш make-файл:

```
dupfind: $(SRC)
    java -Xmx512m -jar ./dupfind.jar > output/dupfind.out
```

Затем добавьте цель dupfind в вашу цель `target`. Осталось только настроить сам **dupfind**. Для этого создайте файл `dupfind.cfg` в текущем каталоге:

```
{
   min: 30,
   max: 500,
   increment: 10,
   fuzzy: true,
   cpd: true,
   sources:
   [
      {
         name: "myProject",
         def: true,
         root: ".",
         directories:
         [
           "src"
         ],
         include:
         [
           "*.js"
         ],
         exclude:
         [
           "*/.svn",
           "*-[^/]*.js",
         ]
      }
   ]
}
```

Это конфигурационный файл в формате JSON, говорящий dupfind, что нужно сделать. Самое интересное в нем — это свойство `fuzzy`, позволяющее **dupfind** делать нечеткие действия (например, игнорировать имена переменных), и массив `sources`, определяющий, какие каталоги и какие файлы нужно исследовать, а какие нужно проигнорировать.

Как только у вас есть такой файл конфигурации, ваша панель инструментов отобразит график дублирующегося кода, как показа-

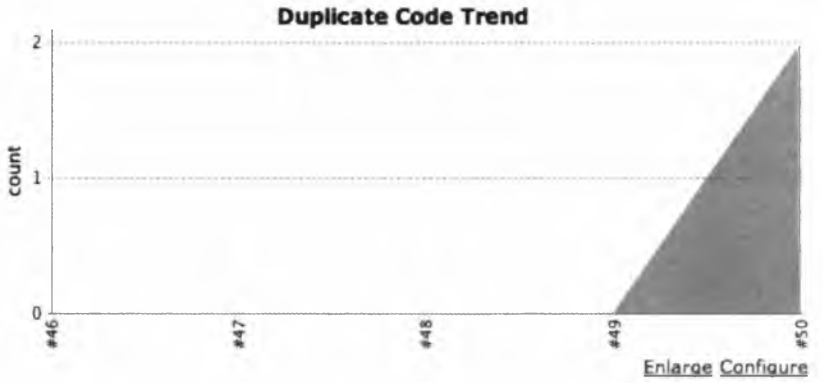


Рис. 8.22. График дублирующегося кода

но рис. 8.22. Если щелкнуть по нему, вы увидите подробности (рис. 8.23) – детальный отчет о дублирующемся коде, говорящий о том, в каких файлах дублируется код и сколько строк кода дублируется.

Рис. 8.23. Отчет о дублировании кода

Уведомления

Jenkins не должен работать сам в себе, как в вакууме — в этом нет никакой необходимости. Настройте действие **E-mail Notification** и по окончании сборки разным группам пользователей будут отправлены соответствующие уведомления по электронной почте.

В сообщениях будет указано состояние сборки, которое может быть:

- "successful" (успех),
- "fail" (неудача),
- "unstable" (нестабильная сборка).

Jenkins различает неудачную и нестабильную сборку: сборка считается неудачной, когда одна из команд сборки вернет ненулевое значение. Нестабильная сборка — когда один из плагинов сообщит, что превышено (или, наоборот, не достигнуто) какое-то допустимое значение, например, покрытие кода слишком низкое.

Поскольку сборка занимает много времени, я настоятельно рекомендую включить эту функцию.

Зависимые сборки

Вы можете запустить создание другой Jenkins-сборки, если текущая сборка была успешно завершена. Также можно настраивать различные действия над текущей сборкой после ее создания. Все это увеличивает автоматизацию процесса разработки.

Ничего лишнего

Мы обсудили довольно много полезных плагинов для Jenkins, и существует еще масса других. При этом не должно быть ничего лишнего: только то, что вы реально используете. На рис. 8.24 показан внешний вид панели инструментов Jenkins со всеми вышеупомянутыми плагинами и ничего лишнего.

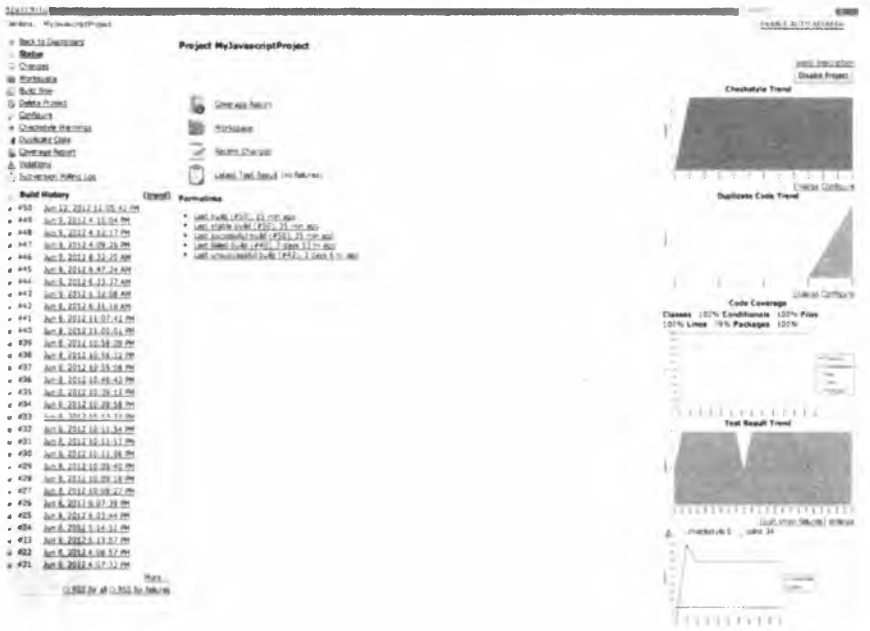


Рис. 8.24. Полностью настроенная домашняя страница проекта Jenkins

Работаем без интерфейса пользователя

Вы можете практически полностью управлять Jenkins через командную строку или удаленный API. При этом существует два метода управления Jenkins из командной строки: через SSH или через `cli.jar`. Jenkins запускает свой собственный SSHD-сервер, по умолчанию работающий на случайном порту.

Чтобы определить номер порта, сделайте запрос HEAD к вашему базовому URL Jenkins и найдите ключ заголовка `X-SSHEndpoint`. Значение этого ключа и есть тот самый случайный номер порта, который прослушивает SSHD-сервер Jenkins. В качестве альтернативы, вы можете настроить Jenkins на использование фиксированного порта в разделе **Configure System**.

Мой случайный номер SSH-порта — 64812, поэтому команда подключения к SSHD-серверу будет такой:

```
% ssh -p 64812 localhost build MyJavaScriptProject -s
```

Данная команда начинает сборку проекта `MyJavaScriptProject`. Вам остается только подождать результаты сборки.

Если вы установили `cli.jar`, то вам станет доступным URL http://<ваш_локальный_jenkins>/cli. Для установки `cli.jar` загрузите его с сайта проекта и выполните следующую команду (все в одной строке):

```
% java -jar ~/Downloads/jenkins-cli.jar -s http://localhost:8080 build
MyJavaScriptProject -s
```

В данном случае начинается сборка проекта `MyJavaScriptProject` и ожидание ее завершения. По завершении сборки вы получите состояние сборки.

Примечание.

Вы можете практически полностью управлять Jenkins через cli.jar. Для дополнительной информации посетите адрес http://<ваш_локальный_jenkins>/cli. Нужно отметить, что через SSH доступна только часть команд, которые вы можете выполнить через cli.jar.

8.3.4. Развертывание

После того, как вы убедитесь, что ваш код хорош, можно выпускать его в производство, то есть разворачивать его в рабочее состояние.

Если код уже был развернут на сервере (старая или тестовая версия), вам нужно корректно завершить работу старого кода и заменить его новым кодом. В данной ситуации первая наша цель — временно прервать (отбросить) как можно меньше соединений (если таковые имеются). У большинства веб-серверов есть «корректная» функция перезапуска, которой и нужно воспользоваться. Если у вас есть только один сервер, то все соединения будут оборваны, пока сервер перезагружается, и это может быть критичным. Если же у вас 50 000 серверов и множество различных балансировщиков нагрузки, то процесс перезапуска одного сервера будет практически незаметным.

На фоне развертывания должна быть доступна функция отката. Вы должны быть готовыми переключиться назад на старый код как

можно быстрее — на случай, если что-то пойдет не так. В идеале, желательно иметь специальную символическую ссылку, позволяющую быстро переключаться между старым и новым кодом. При этом на время обновления оставьте в системе старый код — он может вам пригодиться.

Общая последовательность действий при обновлении кода может выглядеть так:

1. **Загрузка новых файлов на хост.**
2. **Останов сервера.**
3. **Обновление символической ссылки так, чтобы она указывала на новый код.**
4. **Запуск сервера.**
5. **Использование!**

Наверное надо подробнее рассказать о символической ссылке, позволяющей переключаться между новым и старым кодом. Представим, что `DocumentRoot` в вашем конфигурационном файле сервера Apache указывает на символическую ссылку `/var/www/current`, которая разрешается в `/var/www/VERSION_X`. Новый код поместите в каталог `/var/www/VERSION_Y`. Затем остановите Apache и измените символическую ссылку `current` с `VERSION_X` на `VERSION_Y`.

Когда понадобится откат, просто остановите сервер и поменяйте символическую ссылку обратно на `VERSION_X` и заново запустите сервер. И конечно же, можно написать сценарий, который будет выполнять это действие автоматически.

Резюме

Автоматизация процессов разработки, сборки и развертывания крайне важны для нормального, эффективного жизненного цикла приложения. К счастью, есть много JavaScript-инструментов, которые все вместе могут помочь вам с этим. Объединение этих инструментов (от ловушек предварительной фиксации до плагинов Jenkins), позволит вам существенно облегчить себе жизнь. В то же время, любой инструмент приносит пользу только тогда, когда используется. Поэтому должна быть определенная дисциплина применения всех рассмотренных нами инструментов.

Издательство «Наука и Техника»

КНИГИ ПО КОМПЬЮТЕРНЫМ ТЕХНОЛОГИЯМ, МЕДИЦИНЕ, РАДИОЭЛЕКТРОНИКЕ

Уважаемые читатели!

Книги издательства «Наука и Техника» вы можете:

➢ **заказать в нашем интернет-магазине БЕЗ ПРЕДОПЛАТЫ по ОПТОВЫМ ценам**

www.nit.com.ru

- более 3000 пунктов выдачи на территории РФ, доставка 3—5 дней
- более 300 пунктов выдачи в Санкт-Петербурге и Москве, доставка — на следующий день

Справки и заказ:

- на сайте **www.nit.com.ru**
- по тел. (812) 412-70-26
- по эл. почте nitmail@nit.com.ru

➢ **приобрести в магазине издательства по адресу:**

Санкт-Петербург, пр. Обуховской обороны, д.107
М. Елизаровская, 200 м за ДК им. Крупской
Ежедневно с 10.00 до 18.30
Справки и заказ: тел. (812) 412-70-26

➢ **приобрести в Москве:**

«Новый книжный» Сеть магазинов ТД «БИБЛИО-ГЛОБУС»	тел. (495) 937-85-81, (499) 177-22-11 ул. Мясницкая, д. 6/3, стр. 1, ст. М «Лубянка» тел. (495) 781-19-00, 624-46-80
Московский Дом Книги, «ДК на Новом Арбате»	ул. Новый Арбат, 8, ст. М «Арбатская», тел. (495) 789-35-91
Московский Дом Книги, «Дом технической книги»	Ленинский пр., д.40, ст. М «Ленинский пр.», тел. (499) 137-60-19
Московский Дом Книги, «Дом медицинской книги»	Комсомольский пр., д. 25, ст. М «Фрунзенская», тел. (499) 245-39-27
Дом книги «Молодая гвардия»	ул. Б. Полянка, д. 28, стр. 1, ст. М «Полянка» тел. (499) 238-50-01

➢ **приобрести в Санкт-Петербурге:**

Санкт-Петербургский Дом Книги	Невский пр. 28, тел. (812) 448-23-57
Буквоед. Сеть магазинов	тел. (812) 601-0-601

➢ **приобрести в регионах России:**

г. Воронеж, «Амиталь» Сеть магазинов	тел. (473) 224-24-90
г. Екатеринбург, «Дом книги» Сеть магазинов	тел. (343) 289-40-45
г. Нижний Новгород, «Дом книги» Сеть магазинов	тел. (831) 246-22-92
г. Владивосток, «Дом книги» Сеть магазинов	тел. (423) 263-10-54
г. Иркутск, «Продалить» Сеть магазинов	тел. (395) 298-88-82
г. Омск, «Техническая книга» ул. Пушкина, д.101	тел. (381) 230-13-64

Мы рады сотрудничеству с Вами!

Кириченко Андрей Валентинович

JavaScript для FrontEnd-разработчиков

Написание. Тестирование. Развертывание

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО «Наука и Техника»
Лицензия №000350 от 23 декабря 1999 года.
192029, г. Санкт-Петербург, пр.Обуховской обороны, д. 107.
Подписано в печать 30.09.2019. Формат 70x100 1/16.
Бумага офсетная. Печать офсетная. Объем 20 п. л.
Тираж 1200. Заказ 9999.

Отпечатано с готового оригинал-макета
ООО «Принт-М», 142300, М.О., г.Чехов, ул. Полиграфистов, д.1

Кириченко А. В.

JavaScript

для FrontEnd-разработчиков

Написание. Тестирование. Развертывание

Данная книга посвящена тому, как на языке JavaScript создавать хороший код для фронтенда (и не только). В книге последовательно затронуты все аспекты производства JavaScript-кода: от выбора архитектуры и конструирования кода до покрытия модульными тестами, отладки, интеграционного тестирования, сборки и непрерывной поставки вашего кода. Рассматриваются как общие моменты – постановка процесса разработки, событийно-ориентированная архитектура JavaScript-приложений, техника непрерывной интеграции, так и предельно конкретные вопросы – как и какие инструменты (фреймворки) использовать для той или иной задачи, что конкретное нужно делать в том или ином случае, какие ошибки встречаются. Попутно в книге рассмотрено применение большого количества инструментов. Существенное внимание уделено автоматизации на всех этапах создания и поставки JavaScript-кода.

Книга написана доступным языком и представляет несомненный интерес для всех, кто занимается или планирует заняться программированием на JavaScript, хочет повысить качество своего JavaScript-кода, добиться высокой эффективности в создании качественного кода фронтенда. Книга будет полезна как начинающим, так и опытным JavaScript-разработчикам.

www.nit.com.ru

ISBN 978-5-94387-789-6

Издательство "Наука и Техника"
г. Санкт-Петербург